# 8 GIUGNO 2016

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 4) che rappresentano i primi N nomi di file (F0, F1, ... FN-1), mentre l'ultimo rappresenta un numero intero (H) strettamente positivo e minore di 255 (da controllare) che indica la lunghezza in linee dei file: infatti, la lunghezza in linee dei file è la stessa (questo viene garantito dalla parte shell e NON deve essere controllato). Il processo padre deve, per prima cosa, inizializzare il seme per la generazione random di numeri (come illustrato nel retro del foglio) e deve creare un file di nome "/tmp/creato" (Fcreato). Il processo padre deve, quindi, generare N processi figli (P0 ... PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file Fk (con k= i+1). Ogni processo figlio Pi deve leggere le linee del file associato Fk sempre fino alla fine. I processi figli Pi e il processo padre devono attenersi a questo schema di comunicazione: per ogni linea letta, il figlio Pi deve comunicare al padre la lunghezza della linea corrente compreso il terminatore di linea (come int); il padre usando in modo opportuno la funzione mia_random() (riportata nel retro del foglio) deve individuare in modo random**, appunto, quale lunghezza (come int) considerare fra quelle ricevute, rispettando l'ordine dei file, da tutti i figli Pi; individuata questa lunghezza, usando sempre in modo opportuno la funzione mia_random() deve individuare un intero che rappresenterà un indice per la linea della lunghezza considerata ; il padre deve comunicare indietro a tutti i figli Pi tale indice: ogni figlio Pi ricevuto l'indice (per ogni linea) deve controllare se è ammissibile per la linea corrente e in tal caso deve scrivere il carattere della linea corrente, corrispondente a tale indice, sul file Fcreato, altrimenti non deve fare nulla e deve passare a leggere la linea successiva. Al termine, ogni processo figlio Pi deve ritornare al padre il valore intero corrispondente al numero di caratteri scritti sul file Fcreato (sicuramente minore di 255); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

*Precisazione fatta durante lo svolgimento della prova!
**Precisazione fatta durante lo svolgimento della prova: la funzione mia_random() deve essere chiamata due volte dal padre; la prima volta con il numero di processi (in questo caso N); la prima chiamata serve per individuare il figlio di cui deve essere considerata la lunghezza inviata; la seconda volta con la lunghezza individuata al passo precedente!

**tag**: il padre riceve dai figli un numero di informazioni in numero NOTO, i figli ricevono dal padre un numero di informazioni in numero noto

```c
/* Soluzione della Prova d'esame del 8 Giugno 2016 - Parte C */
/* QUESTA SOLUZIONE NON GESTISCE LATO PADRE IL CASO CHE UN FIGLIO TERMINI PERCHE'
IL FILE ASSOCIATO NON ESISTE (IL FIGLIO PERO' CONTROLLA IL RISULTATO DELLA OPEN E
TERMINA IN CASO DI ERRORE): SI PUO' IPOTIZZARE CHE POSSA ANDARE BENE VISTO CHE LA
PARTE C VIENE INVOCATA DA UNA PARTE SHELL, CHE SE SVOLTA CORRETTAMENTE, INVIA NOMI
DI FILE ESISTENTI */
/* SE SI PROVA QUESTO PROGRAMMA CON UN FILE CHE CON ESISTE IL PADRE AL TENTATIVO DI
SCRIVERE SULLA PIPE DI CONNESSIONE DI QUEL FIGLIO RICEVE UN SIGPIPE E TERMINA */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>

typedef int pipe_t[2];

int mia_random(int n)
{
   int casuale;
        casuale = rand() % n;   /* con rand() viene calcolato un numero pseudo
casuale e con l'operazione modulo n si ricava un numero casuale compreso fra 0 e n-
```

```c
1 */
        return casuale;
}

int main(int argc, char **argv)
{
   /* -------- Variabili locali ---------- */
   int pid;                     /* process identifier per le fork() */
   int N;                       /* numero di file passati sulla riga di comando
(uguale al numero di figli) */
   int H;                       /* numero passato come ultimo parametro e che
rappresenta la lunghezza in linee dei file passati sulla riga di comando */
   int fdout;                   /* file descriptor per creazione file da parte del
padre */
   int fd;                      /* file descriptor per apertura file */
   pipe_t *pipedFP;             /* array dinamico di pipe descriptors per
comunicazioni figli-padre */
   pipe_t *pipedPF;             /* array dinamico di pipe descriptors per
comunicazioni padre-figli */
   int i, j;                    /* indici per i cicli */
   char linea[255];             /* array di caratteri per memorizzare la linea,
supponendo una lunghezza massima di ogni linea di 255 caratteri compreso il
terminatore di linea */
   int valore;                  /* variabile che viene usata dal padre per
recuperare il valore comunicato da ogni figlio e che contiene la lunghezza della
linea corrente */
   int giusto;                  /* variabile che viene usata dal padre per salvare
per ogni linea il valore inviato dal figlio selezionato in modo random */
   int r;                       /* variabile usata dal padre per calcolare i valori
random e dal figlio per ricevere il numero dal padre */
   int ritorno=0;               /* variabile che viene ritornata da ogni figlio al
padre e che contiene il numero di caratteri scritti sul file (supposta minore di
255): valore iniziale 0 */
   int status;                  /* variabile di stato per la wait */
   /* ------------------------------------ */

        /* Controllo sul numero di parametri */
        if (argc < 6) /* Meno di cinque parametri */
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno cinque
parametri (quattro file e un numero intero strettamente positivo e minore di
255!)\n", argc);
                exit(1);
        }

        /* Calcoliamo il numero di file passati */
        N = argc - 2;
        /* convertiamo l'ultima stringa in un numero */
        H = atoi(argv[argc-1]);
        /* controlliamo il numero: deve essere strettamente positivo e minore di
255 */
        if ((H <= 0) || (H >= 255))
        {
                printf("Errore nel numero passato %d\n", H);
                exit(2);
        }

        /* inizializziamo il seme per la generazione random di numeri  */
        srand(time(NULL));
```

```
        /* creiamo il file in /tmp */
        if ((fdout=open("/tmp/creato", O_CREAT|O_WRONLY|O_TRUNC, 0644)) < 0) /*
usato la open in versione estesa per azzerare il file nel caso esista gia' */
        {
                printf("Errore nella creazione del file %s\n", "/tmp/creato");
                exit(3);
        }
        /* NOTA BENE: tutti i figli avranno la possibilita' di accedere in
scrittura al file creato dal padre (quindi non devono fare alcuna open) e avendo il
file pointer condiviso non ci sara' bisogno di spostare l'I/O pointere */

        /* Allocazione dei due array di N pipe descriptors*/
        pipedFP = (pipe_t *) malloc (N*sizeof(pipe_t));
        pipedPF = (pipe_t *) malloc (N*sizeof(pipe_t));
        if ((pipedFP == NULL) || (pipedPF == NULL))
        {
                printf("Errore nella allocazione della memoria per le due serie di
N pipe\n");
                exit(4);
        }

        /* Creazione delle N pipe figli-padre e delle N pipe padre-figli */
        for (i=0; i < N; i++)
        {
                if(pipe(pipedFP[i]) < 0)
                {
                        printf("Errore nella creazione della pipe figli-padre\n");
                        exit(5);
                }
                if(pipe(pipedPF[i]) < 0)
                {
                        printf("Errore nella creazione della pipe padre-figli\n");
                        exit(6);
                }
        }

        printf("DEBUG-Sono il padre con pid %d e sto per generare %d figli\n",
getpid(), N);

        /* Ciclo di generazione dei figli */
        for (i=0; i < N; i++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork\n");
                        exit(7);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        printf("DEBUG-Sono il processo figlio di indice %d e pid %d
e sono associato al file %s\n", i, getpid(), argv[i+1]);
                        /* Chiusura delle pipe non usate nella comunicazione con il
padre */
                        /* ogni figlio scrive solo su pipeFP[i] e legge solo da
pipePF[i] */
```

```
                        for (j=0; j < N; j++)
                        {
                                close(pipedFP[j][0]);
                                close(pipedPF[j][1]);
                                if (j != i)  {
                                                close(pipedFP[j][1]);
                                                close(pipedPF[j][0]);
                                                }
                        }

                        /* apriamo il file associato in sola lettura */
                        if ((fd=open(argv[i+1], O_RDONLY)) < 0)
                        {
                                printf("Errore nella open del file %s\n",
argv[i+1]);
                                exit(-1); /* decidiamo di tornare -1 che verra'
interpretato dal padre come 255 e quindi un valore non ammissibile! */
                                /* OSSERVAZIONE IMPORTANTE: in realta' per come e'
lo schema di comunicazione di questo compito, se un figlio muore in questa fase, il
padre nella fase di invio dell'indice (calcolato in modo random) riceve il segnale
SIGPIPE e quindi se il segnale non viene trattato l'azione di default e' la
terminazione del padre che quindi NON attende i figli e non recupera neanche i
valori di ritorno */
                        }

                        /* adesso il figlio legge dal file una linea alla volta */
                        j=0;    /* riusiamo la variabile j */
                        while (read(fd, &(linea[j]), 1))
                        {
                                if (linea[j] == '\n')
                                {       /* dobbiamo mandare al padre la lunghezza
della linea corrente compreso il terminatore di linea (come int) e quindi
incrementiamo j */
                                        j++;
                                        write(pipedFP[i][1], &j, sizeof(j));
                                        /* il figlio Pi deve leggere il valore
inviato dal padre */
                                        read(pipedPF[i][0], &r, sizeof(r));
                                        /* il figlio Pi deve controllare
l'ammissibilita' del valore inviato dal padre rispetto alla lunghezza della linea
corrente */
                                        if (r < j)
                                        {
                                                /* dato che si riceve un indice che
varia da 0 alla lunghezza - 1 scelta dal padre e dato che j rappresenta la
lunghezza della linea corrente (compreso il terminatore di linea), l'indice
ricevuto per essere ammissibile deve essere strettamente minore di j; N.B. se r
dovesse risultare uguale a j-1 allora il carattere che verrebbe selezionato sarebbe
lo '\n'! */
                                                /* stampa di controllo:
                                                printf("DEBUG-Processo di indice %d
sto per scrivere carattere %c nel file\n", i, linea[r]);
                                                */
                                                /* ogni figlio usa, per scrivere
sul file, il valore di fdout ottenuto per copia dal padre e condivide l'I/O pointer
con il padre e tutti i fratelli */
                                                write(fdout, &(linea[r]), 1);
                                                /* il figlio incrementa il valore
di ritorno */
```

```
                                        ritorno++;
                                }
                                else
                                        /* stampa di controllo:
                                        printf("Processo di indice %d non
scrive nulla nel file\n", i);
                                        */
                                        ; /* non si deve fare nulla */
                                j=0; /* azzeriamo l'indice per le prossime
linee */
                        }
                        else    j++; /* continuiamo a leggere */
                }
                /* il figlio Pi deve ritornare al padre il valore
corrispondente al numero di caratteri scritti sul file */
                exit(ritorno);
        }
}

/* Codice del padre */
/* chiudiamo le pipe che non usiamo: il padre legge da tutte le pipe pipeFP
e scrive su tutte le pipe pipePF */

for (i=0; i < N; i++)
{
        close(pipedFP[i][1]);
        close(pipedPF[i][0]);
}

/* Il padre recupera le informazioni dai figli: prima in ordine di linee e
quindi in ordine di indice */
for (j=1; j <= H; j++)
{
        /* il padre calcola in modo random l'indice del figlio di cui
considerera' il valore inviato */
        r=mia_random(N);
        /* stampa di controllo */
        printf("DEBUG-Il numero generato in modo random (per la linea %d)
compreso fra 0 e %d per selezionare il figlio e' %d\n", j, N-1, r);
        for (i=0; i < N; i++)
        {
                /* il padre recupera tutti i valori interi dai figli */
                read(pipedFP[i][0], &valore, sizeof(valore));
                /* ma si salva solo il valore del figlio identificato in
modo random */
                if (i == r)
                        giusto=valore;
        }
        /* il padre calcola in modo random l'indice della linea che
inviera' a tutti i figli */
        r=mia_random(giusto);
        /* stampa di controllo */
        printf("DEBUG-Il numero generato in modo random compreso fra 0 e %d
per selezionare l'indice della linea %d-esima e' %d\n", giusto-1, j, r);
        /* il padre deve inviare a tutti i figli questo indice */
        for (i=0; i < N; i++)
                write(pipedPF[i][1], &r, sizeof(r));
}
```

```
        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                if ((pid = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(8);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {       ritorno=(int)((status >> 8) &   0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 255 ci
sono stati errori nel figlio)\n", pid, ritorno);
                }
        }

        exit(0);
}
```

# 9 SETTEMBRE 2015

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 2) che rappresentano N+1 nomi di file (F1, F2,... FN e AF) con uguale lunghezza in byte (che viene assicurata dalla parte shell e non si deve controllare). Il processo padre è associato al file AF e deve generare N processi figli (P0 ... PN-1) ognuno dei quali è associato ad uno dei file Fi. Ogni processo figlio Pi deve leggere i caratteri del file associato Fi sempre fino alla fine solo dopo aver ricevuto l'indicazione dal padre di procedere. Infatti, i processi figli devono attenersi a questo schema di comunicazione/sincronizzazione con il padre: il figlio P0, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo confronta con il primo carattere del file AF; il figlio P1, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo confronta con il primo carattere del file AF etc. fino al figlio PN-1, ricevuta l'indicazione dal padre che può procedere, legge il primo carattere e lo comunica al padre che lo confronta con il primo carattere del file AF; questo schema deve continuare per gli altri caratteri solo se il confronto ha successo (cioè i due caratteri sono uguali); se un confronto non ha successo, il padre non deve inviare al corrispondente figlio l'indicazione di procedere. Una volta che il padre termina la lettura del file associato AF, deve terminare forzatamente (con un apposito segnale) i figli per i quali il confronto non abbia avuto successo. Al termine, i processi figli Pi che non sono stati terminati forzatamente devono ritornare al padre l'indicazione di successo secondo la semantica di UNIX; il padre deve stampare su standard output il PID di ogni figlio con l'indicazione di terminazione anormale o normale e in questo caso il valore ritornato dal figlio insieme con il nome del file il cui contenuto risulta uguale a quello del file AF.

**tag**: il padre invia ai figli se possono procedere. I figli inviano al padre informazioni in numero NON noto, se i figli terminano il padre invia il segnale `SIGKILL`

```
/* versione con N pipe di comunicazione/sincronizzazione fra padre e figli e altre
N di comunicazione fra figli e padre; inoltre uso del segnale SIGKILL e del segnale
SIGUSR1 */
/* SI CONSIDERI CHE IL PADRE, SE UN FIGLIO TERMINA PERCHE' IL PROPRIO FILE NON
ESISTE, QUANDO TENTA DI SCRIVERE A QUEL FIGLIO RICEVE UN SIGPIPE: SI CONSIDERI IL
CODICE CHE SI DOVREBBE AGGIUNGERE PER TRATTARE QUESTO CASO PER INTERCETTARE QUESTO
CASO */
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>

typedef int pipe_t[2];

/* MESSO GLOBALE IN MODO DA POTERLO USARE NELLA FUNZIONE HANDLER, anche se la stampa e' solo di debugging */
int i;

void handler(int signo)
{
        printf("DEBUG-Sono il figlio %d di indice %d e ho finito di leggere il mio file\n", getpid(), i);
        exit(0); /* il figlio termina in modo normale */
}

int main (int argc, char **argv)
{
   int N;                               /* numero di file */
   int *pid;                            /* array di pid per fork: N.B. in questo caso serve un array di pid perche' il padre deve inviare il segnale SIGKILL per terminare forzatamente i figli per i quali il confronto non abbia avuto successo; in questa soluzione poi agli altri viene mandato il segnale SIGUSR1! */
   int *confronto;                      /* array dinamico di interi per sapere se si deve ancora mandare l'indicazione di leggere al figlio corrispondente */
   pipe_t *pipeFiglioPadre;             /* array di pipe di comunicazione fra figli e padre */
   pipe_t *pipePadreFiglio;             /* array di pipe di comunicazione/sincronizzazione fra padre e figlio. NOTA BENE: questa sincronizzazione potrebbe essere fatta tramite l'invio di segnali da parte del padre ai figli */
   int j;                               /* indice per i cicli */
   int fd;                              /* file descriptor */
   char c,ch;                           /* carattere per leggere dal file da parte dei figli e dalle pipe da parte del padre*/
   char token='v';                      /* carattere che serve per sincronizzare i figli: in questo caso in realta' non importa il valore del carattere! */
   int pidFiglio, status, ritorno;      /* per valore di ritorno figli */

        /* ci vogliono almeno due file per i figli che insieme con AF fa un totale di almeno 3 parametri */
        if (argc < 4)
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno tre parametri (due file usati dai figli e in file usato dal padre)\n", argc);
                exit(1);
        }

        N = argc-2; /* calcoliamo il numero di file passati, a parte AF, e quindi il numero di figli da creare */
        printf("DEBUG-Sono il padre %d e il numero di processi da creare sono %d\n", getpid(), N);

        /* allocazione array pid */
```

```
        if ((pid=(int *)malloc(N*sizeof(int))) == NULL)
        {
                printf("Errore allocazione pid\n");
                exit(2);
        }

        /* allocazione array confronto */
        if ((confronto=(int *)malloc(N*sizeof(int))) == NULL)
        {
                printf("Errore allocazione confronto\n");
                exit(3);
        }

        /* inizializzazione di confronto: tutti i valori a 1 perche' all'inizio si
deve mandare l'indicazione al figlio di leggere! */
        for (i=0;i<N;i++)
                confronto[i]=1;

        /* allocazione pipe figli-padre */
        if ((pipeFiglioPadre=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe padre\n");
                exit(4);
        }

        /* allocazione pipe padre-figli */
        if ((pipePadreFiglio=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe padre\n");
                exit(5);
        }

        /* Creazione delle N pipe figli-padre e delle N pipe padre-figli */
        for (i=0;i<N;i++)
        {
                if(pipe(pipeFiglioPadre[i])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(6);
                }

                /* N.B. invece di queste N pipe il padre, si potrebbe usare lo
strumento dei segnali */
                if(pipe(pipePadreFiglio[i])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(7);
                }
        }

        /* il padre installa il gestore handler per il segnale SIGUSR1: questa
installazione serve per i figli! */
        signal(SIGUSR1, handler);

        /* creazione figli con salvataggio dei pid nell'array */
        for (i=0;i<N;i++){
                if ((pid[i]=fork())<0)
                {
                        printf("Errore creazione figli\n");
```

```
                        exit(8);
                }
                if (pid[i]==0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d di indice %d\n", getpid(),
i);

                        /* chiusura pipe inutilizzate */
                        for (j=0;j<N;j++)
                        {
                                /* il figlio non legge dai nessuna pipeFiglioPadre
e non scrive su nessuna pipePadreFiglio */
                                close (pipeFiglioPadre[j][0]);
                                close (pipePadreFiglio[j][1]);
                                if (j != i)
                                {       /* inoltre non scrive e non legge se non
su/dalle sue pipe */
                                        close (pipeFiglioPadre[j][1]);
                                        close (pipePadreFiglio[j][0]);
                                }
                        }

                        /* apertura file */
                        if ((fd=open(argv[i+1],O_RDONLY)) < 0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[i+1]);
                                exit(-1); /* deciso di tornare -1 che verra'
interpretato come 255 e quindi INSUCCESSO */
                        }

                        /* ATTENZIONE LO SCHEMA DA UTILIZZARE E' DIVERSO DA QUELLO
DEL 15 LUGLIO 2015 DATO CHE IN QUEL CASO SI DEVE USCIRE DAL CICLO APPENA FINISCE IL
PROPRIO FILE ASSOCIATO; INVECE IN QUESTO CASO SI DEVE TERMINARE QUANDO LO
STABILISCE IL PADRE! */
                        /* ciclo di lettura da pipe per ricevere l'indicazione dal
padre e quindi lettura dal file */
                        while (read(pipePadreFiglio[i][0], &token, 1))
                        {
                                read(fd,&c,1);
                                /* printf("DEBUG-HO LETTO IL TOKEN per il carattere
%c\n", c); */

                                /* il carattere letto va mandato al padre per il
confronto */
                                write(pipeFiglioPadre[i][1],&c,1);
                        }
                        exit(0); //NON SI ESEGUIRA' MAI QUESTA EXIT!
                }
        } /* fine for */

        /* codice del padre */
        /* chiusura pipe */
        for(i=0;i<N;i++)
        { /* il padre non legge da nessuna pipePadreFiglio e non scrive su nessuna
pipeFiglioPadre */
                close (pipePadreFiglio[i][0]);
                close (pipeFiglioPadre[i][1]);
        }
```

```
        /* apertura file assegnato al padre (AF) */
        if ((fd=open(argv[argc-1],O_RDONLY)) < 0)
        {
                printf("Impossibile aprire il file %s\n", argv[argc-1]);
                exit(9);
        }

        /* il padre deve leggere i caratteri dal file assegnato e confrontarli con
quelli inviati dai figli: si ricorda che i file hanno tutti la stessa lunghezza
dato che questo e' garantito dalla parte shell */
        while (read(fd, &ch, 1))
                for(i=0;i<N;i++)
                {
                        /* il padre manda l'indicazione di leggere ad ogni figlio
per ogni carattere solo se confronto e' ancora 1 */
                        if (confronto[i])
                        {
                                write(pipePadreFiglio[i][1], &token, 1);
                                /* il padre riceve il carattere letto dal figlio e
lo confronta con il suo */
                                read(pipeFiglioPadre[i][0],&c,1);
                                /* printf("DEBUG-SONO IL PADRE: HO LETTO il
carattere %c e il figlio %c\n", ch, c);  */
                                if (ch != c)
                                {
                                        /* i caratteri sono diversi e quindi
bisogna resettare il valore corrispondente di confronto  */
                                        /* printf("DEBUG-SONO IL PADRE: HO LETTO il
carattere %c e il figlio %c e quindi mi segno che non devo piu'ù mandare
indicazione a questo figlio %d\n", c, c, pid[i]);  */

                                        confronto[i]=0;
                                }
                        }
                }

        /* una volta che il padre ha terminato la lettura del file AF */
        for(i=0;i<N;i++)
                if (!confronto[i])
                {
                        /* terminiamo forzatamente tutti i figli che hanno fallito
il confronto e che sono bloccati sulla read dalla pipe con il padre */
                        if (kill(pid[i], SIGKILL) == -1) /* controlliamo che la
kill non fallisca a causa della terminazione di uno dei figli, anche se in questo
caso non dovremmo avere mai questo caso */
                        printf("Figlio con pid %d non esiste e quindi e' gia'
terminato\n", pid[i]);;
                }
                else
                {
                        /* per i figli invece per cui il confronto non e' fallito
mandiamo il segnale SIGUSR1 in modo che forziamo di uscire dal ciclo e questi figli
termineranno in modo normale, altrimenti rimarrebbero bloccati sulla read e avremmo
un deadlock: dato che il figlio aspetterebbe un token dal padre e il padre
aspetterebbe che il figlio finisse (con la wait del codice seguente) */
                        kill (pid[i], SIGUSR1);
                }

        /* Il padre aspetta i figli */
```

```
        for (i=0; i < N; i++)
        {
                if ((pidFiglio = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(10);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {       ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato il valore %d (se
255 problemi)\n", pidFiglio, ritorno);
                        /* se un figlio termina normalmente vuol dire che non e'
stato ucciso dal SIGKILL: ATTENZIONE CHE DOBBIAMO RECUPERARE L'INDICE DI CREAZIONE
USANDO L'ARRAY DI pid! */
                        for (j=0; j < N; j++)
                                if (pid[j] == pidFiglio)
                                        printf("Questo significa che il figlio di
indice %d ha verificato che il file %s e' uguale al file %s\n", j, argv[j+1],
argv[argc-1]);
                }
        }

        exit(0);
}
```

# 14 GIUGNO 2017

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 1, da controllare) che rappresentano i primi N nomi assoluti di file (F1, F2, ... FN) mentre l'ultimo rappresenta un singolo carattere (Cx) (da controllare): si può ipotizzare che la lunghezza di tutti i file sia uguale (senza verificarlo). Il processo padre deve generare N processi figli (P0, P1, … PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file FK (con K= i+1). Ogni processo figlio Pi deve leggere i caratteri del file associato FK cercando le occorrenze del carattere Cx, sostituendole eventualmente con i caratteri inviati dal padre. Ogni figlio Pi, per ogni occorrenza trovata, deve comunicare al padre la posizione (in termini di long int) di tale occorrenza a partire dall'inizio del file1* . Il padre deve ricevere le posizioni (come long int) inviate dai figli nel seguente ordine: prima deve ricevere dal figlio P0 la prima posizione inviata, poi deve ricevere dal figlio P1 la prima posizione inviata e così via fino a che deve ricevere dal figlio PN-1 la prima posizione inviata; quindi deve procedere a ricevere le seconde posizioni inviate dai figli (se esistono) e così via. La ricezione di posizioni da parte del padre deve terminare quando ha ricevuto tutte le posizioni inviate da tutti i figli Pi. Per ogni posizione ricevuta, il padre deve riportare sullo standard output: l'indice del figlio che gli ha inviato la posizione, il nome del file in cui è stata trovata l'occorrenza del carattere Cx e (naturalmente) la posizione ricevuta. Quindi, per ogni posizione ricevuta, il padre deve chiedere all'utente il carattere con cui deve essere sostituita la specifica occorrenza; nel caso l'utente inserisca una linea vuota, questo deve essere interpretato dal padre come indicazione di non sostituire l'occorrenza corrente. Il padre, per ogni posizione, deve comunicare al figlio corrispondente o il carattere da sostituire oppure se può proseguire con la ricerca di altre occorrenze del carattere Cx. Al termine, ogni processo figlio Pi deve ritornare al padre il numero di sostituzioni effettuate nel proprio file (supposto strettamente minore di 255); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

1* Si precisi, come commento, nel codice se il primo carattere del file viene considerato in posizione 0 o in posizione 1.

**tag**: i figli inviano un numero non noto di posizioni del carattere Cx cercato al padre, il padre invia un carattere da sostituire ai figli.

```c
/* Soluzione della Prova d'esame del 14 Giugno 2017 - Parte C */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

typedef int pipe_t[2];

int main(int argc, char **argv)
{
        /* -------- Variabili locali ---------- */
        int pid;                        /* process identifier per le fork() */
        int N;                          /* numero di file passati sulla riga di
comando */
        char Cx;                        /* carattere passato come ultimo parametro
*/
        int fd;                         /* file descriptor per apertura file */
        long int pos;                   /* valore per comunicare la posizione al
padre e quindi all'utente */
        int status;                     /* variabile di stato per la wait */
        pipe_t *pipedFP;                /* array dinamico di pipe descriptors per
comunicazioni figli-padre */
        pipe_t *pipedPF;                /* array dinamico di pipe descriptors per
comunicazioni padre-figli */
        int i, j;                       /* indici per i cicli */
        char cx;                        /* variabile che viene usata dal padre per
avere dall'utente l'informazione del carattere da sostituire e  dai figli per
recuperare il carattere comunicato dal padre */
        char scarto;                    /* variabile che viene usata dal padre per
eliminare lo '\n' letto dallo standard input */
        char ch;                        /* variabile che viene usata dai figli per
leggere dal file */
        int ritorno=0;                  /* variabile che viene ritornata da ogni
figlio al padre e che contiene il numero di caratteri sostituiti nel file (supposto
minore di 255 */
        int nr;                         /* variabile che serve al padre per sapere
se non ha letto nulla */
        int finito;                     /* variabile che serve al padre per sapere
se non ci sono piu' posizioni da leggere */

        /* ---------------------------------- */

        /* Controllo sul numero di parametri */
        if (argc < 3) /* Meno di due parametri: ci vuole almeno un file e il
carattere da cercare */
        {
                printf("Errore nel numero dei parametri dato che argc=%d (ce ne
vogliono almeno due, un file e un carattere)\n", argc);
                exit(1);
        }
        /* Calcoliamo il numero di parametri passati */
```

```
        N = argc - 2;
        /* Isoliamo il carattere, dopo aver controllato che sia un singolo
carattere */
        if (argv[argc-1][1] != '\0')
        {
                printf("Errore nell'ultimo parametro %s: non singolo carattere\n",
argv[argc-1]);
                exit(2);
        }
        Cx = argv[argc-1][0];

        printf("DEBUG-Sono il padre con pid %d e creo %d figli che dovranno cercare
il carattere %c\n", getpid(), N, Cx);

        /* Allocazione dei due array di N pipe descriptors*/
        pipedFP = (pipe_t *) malloc (N*sizeof(pipe_t));
        pipedPF = (pipe_t *) malloc (N*sizeof(pipe_t));
        if ((pipedFP == NULL) || (pipedPF == NULL))
        {
                printf("Errore nella allocazione della memoria per le pipe\n");
                exit(3);
        }

        /* Creazione delle N pipe figli-padre e delle N pipe padre-figli */
        for (i=0; i < N; i++)
        {
                if(pipe(pipedFP[i]) < 0)
                {
                        printf("Errore nella creazione della pipe figli-padre\n");
                        exit(4);
                }
                if(pipe(pipedPF[i]) < 0)
                {
                        printf("Errore nella creazione della pipe padre-figli\n");
                        exit(5);
                }

        }

/* Ciclo di generazione dei figli */
        for (i=0; i < N; i++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork\n");
                        exit(6);
                }

                if (pid == 0)
                {
                        /* codice del figlio: in caso di errore torniamo -1 che
verra' considerato come 255 dal padre */
                        printf("DEBUG-Sono il processo figlio di indice %d e pid %d
e sono associato al file %s\n", i, getpid(), argv[i+1]);
                        /* Chiusura delle pipe non usate nella comunicazione con il
padre */
                        for (j=0; j < N; j++)
                        {
                                close(pipedFP[j][0]);
```

```
                                close(pipedPF[j][1]);
                                if (i != j)  {
                                                close(pipedFP[j][1]);
                                                close(pipedPF[j][0]);
                                                }
                        }
                        /* apertura del file associato sia in lettura che in
scrittura! */
                        if ((fd=open(argv[i+1], O_RDWR)) < 0)
                        {
                                printf("Errore nella open del file %s\n",
argv[i+1]);
                                exit(-1);
                        }
                        while (read(fd, &ch, 1))
                        {
                                if (ch == Cx)
                                { /* il figlio ha trovato il carattere cercato e
quindi bisogna comunicare al padre la posizione: si puo' fare, ad esempio, usando
la funzione lseek; consideriamo le posizioni dei caratteri a partire dalla
posizione 1 (dato che abbiamo gia' letto il carattere il valore restituito dalla
lseek sara' gia' quello giusto) */
                                        pos = lseek(fd, 0L, SEEK_CUR);
                                        write(pipedFP[i][1], &pos, sizeof(pos));
                                        /* il figlio ora deve aspettare il
carattere da sostituire, se '\n' non deve fare nulla */
                                        read(pipedPF[i][0], &cx, 1);
                                        /* printf("FIGLIO RICEVUTO %c\n", cx); */
                                        /* adesso il figlio deve tornare indietro
con il file pointer per sovrascrivere, nel caso, il carattere trovato con quello
comunicato dal padre */
                                        if (cx != '\n')
                                        {
                                /* per poter sovrascrivere il carattere, come
richiesto dal padre, bisogna tornare indietro di una posizione */
                                                lseek(fd, -1L, SEEK_CUR);
                                                write(fd, &cx, 1);
                                                /* il figlio incrementa il valore
di ritorno, dato che e' stata fatta una sostituzione */
                                                ritorno++;
                                        }
                                }
                        }
                        /* il figlio Pi deve ritornare al padre il valore
corrispondente al numero di sostituzioni effettuate (che si puo' supporre
strettamente minore di 255, come indicato nel testo) */
                        exit(ritorno);
                }
        }

/* Codice del padre */
/* Il padre chiude i lati delle pipe che non usa */
        for (i=0; i < N; i++)
        {
                close(pipedFP[i][1]);
                close(pipedPF[i][0]);
        }

/* Il padre recupera le informazioni dai figli: prima in ordine di posizioni e
```

```
quindi in ordine di indice */
        finito = 0; /* all'inizio supponiamo che non sia finito nessun figlio */
        while (!finito)
        {
                finito = 1; /* mettiamo finito a 1 perche' potrebbero essere finiti
tutti i figli */
                for (i=0; i<N; i++)
                {
                 /* si legge la posizione inviata  dal figlio i-esimo */
                        nr = read(pipedFP[i][0], &pos, sizeof(pos));
                        if (nr != 0)
                        {
                                finito = 0; /* almeno un processo figlio non e'
finito */
                                printf("Il figlio di indice %d ha letto dal file %s
nella posizione %ld il carattere %c. Se vuoi sostituirlo, fornisci il carattere
altrimenti una linea vuota?\n", i,  argv[i+1], pos, Cx);
                                read(0, &cx, 1);
                                if (cx != '\n') read(0, &scarto, 1); /* se e' stato
letto un carattere, bisogna fare una lettura a vuoto per eliminare il carattere
corrispondente all'invio */
                                write(pipedPF[i][1], &cx, 1);  /* inviamo comunque
al figlio */

                        }
                }
         }

        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                pid = wait(&status);
                if (pid < 0)
                {
                        printf("Errore in wait\n");
                        exit(7);
                }
                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {       ritorno=(int)((status >> 8) &   0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi nel figlio)\n", pid, ritorno);
                }
        }

        exit(0);
}
```

# 12 LUGLIO 2017

La parte in C accetta un numero variabile pari 2N di parametri maggiore o uguale a 2 (da controllare) che rappresentano N nomi assoluti di file F1, ... FN intervallati da numeri interi strettamente positivi X1, X2, ... XN che rappresentano la lunghezza in linee dei file (si può supporre che i parametri di posizione pari siano numeri e si deve solo controllare che siano strettamente

positivi). Il processo padre deve generare N processi figli: i processi figli Pi sono associati agli N file Fh e al numero Xh (con h= i+1). Ognuno di tali figli deve creare a sua volta un processo nipote PPi: ogni processo nipote PPi esegue concorrentemente e deve, per prima cosa, inizializzare il seme per la generazione random di numeri (come illustrato nel retro del foglio), quindi deve, usando in modo opportuno la funzione mia_random() (riportata sul retro del foglio), individuare un intero che rappresenterà il numero di linee del file Fh da selezionare, a partire dall'inizio del file, e inviare al figlio usando in modo opportuno il comando head di UNIX/Linux. Ogni processo figlio Pi deve ricevere tutte le linee inviate dal suo processo nipote PPi (ogni linea si può supporre che abbia una lunghezza massima di 250 caratteri, compreso il terminatore di linea e, se serve, il terminatore di stringa) e, per ogni linea ricevuta, deve inviare al processo padre una struttura dati, che deve contenere tre campi: 1) c1, di tipo int, che deve contenere il pid del nipote; c2, di tipo int, che deve contenere il numero della linea; 3) c3, di tipo char[250], che deve contenere la linea corrente ricevuta dal nipote. Il padre deve ricevere le strutture inviate dai figli nel seguente ordine: prima deve ricevere dal figlio P0 la prima struttura inviata, poi deve ricevere dal figlio P1 la prima struttura inviata e così via fino a che deve ricevere dal figlio PN-1 la prima struttura inviata; quindi deve procedere a ricevere le seconde strutture inviate dai figli (se esistono) e così via. La ricezione di strutture da parte del padre deve terminare quando ha ricevuto tutte le strutture inviate da tutti i figli Pi. Il padre deve stampare su standard output, per ogni struttura ricevuta, ognuno dei campi. Al termine, ogni processo figlio Pi deve ritornare al padre il valore random calcolato dal proprio processo nipote PPi e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tag**: i figli inviano un numero non noto informazioni, ma il padre non rimanda nulla indietro.

```c
/* Soluzione della Prova d'esame del 12 Luglio 2017 - Parte C */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>

typedef int pipe_t[2];
typedef struct {
        int pid_nipote;         /* campo c1 del testo */
        int numero_linea;       /* campo c2 del testo */
        char linea_letta[250];  /* campo c3 del testo */
                                /* bastano 250 caratteri per contenere ogni riga
insieme con il terminatore di linea e con il terminatore di stringa (vedi
specifica) */
                } strut;

int mia_random(int n)
{       /* funzione che calcola un numero random compreso fra 1 e n (fornita nel
testo) */
   int casuale;
        casuale = rand() % n;
        casuale++;
        return casuale;
}

int main(int argc, char **argv)
{
   /* -------- Variabili locali ---------- */
   int pid;                     /* process identifier per le fork() */
   int N;                       /* numero di file passati sulla riga di comando
(uguale al numero di numeri), che corrispondera' al numero di figli  */
   pipe_t *piped;               /* array dinamico di pipe descriptors per
```

```c
comunicazioni figli-padre  */
   pipe_t p;                    /* una sola pipe per ogni coppia figlio-nipote */
   int i, j;                    /* indici per i cicli */
   int X, r;                    /* variabili usate dal nipote per il numero di
linee (sicuramente strettamente minore di 255, garantito dalla parte shell) e per
il valore random (anche lui < di 255) */
   int nr;                      /* variabile usata dal padre per controllo sulla
read dai figli */
   char opzione[5];             /* variabile stringa che serve per passare al
comando head l'opzione con il numero delle linee da stampare */
                                /* supponiamo che bastino 3 cifre per il numero,
oltre al simbolo di opzione '-' e al terminatore di stringa */
   strut S;                     /* struttura usata dai figli e dal padre */
   int finito;                  /* variabile che serve al padre per sapere se non
ci sono piu' strutture da leggere */
   int ritorno;                 /* variabile che viene ritornata da ogni figlio al
padre e che contiene il numero random calcolato dal nipote */
   int status;                  /* variabile di stato per la wait */
   /* ---------------------------------- */

        /* Controllo sul numero di parametri */
        if (argc < 3 || (argc-1)%2) /* Meno di due parametri e non pari */
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno due
parametri (o comunque un numero pari di parametri)\n", argc);
                exit(1);
        }

        /* Calcoliamo il numero di file/numeri passati e quindi di figli da creare
*/
        N = (argc - 1)/2;

        /* Controlliamo i numeri passati: non usiamo ora la funzione atoi() per
trasformare queste stringhe in numeri, dato che i numeri saranno convertiti dai
nipoti. Quindi controlliamo il primo carattere (quello di indice 0) di ogni stringa
numerica e se e' il carattere '-' oppure '0' vuol dire che il numero non e'
strettamente positivo e diamo errore */
        for (i=0; i < N; i++)
        {
        /* le stringhe che rappresentano i numeri si trovano con l'indice i
moltiplicato per 2 e sommando 2, cioe' nelle posizioni 2, 4, 6, etc. e quindi
saltando le stringhe che rappresentano i nomi dei file */
                if (argv[(i*2)+2][0] == '-'  || argv[(i*2)+2][0] == '0')
                {
                        printf("Errore %s non rappresenta un numero strettamente
positivo \n", argv[(i*2)+2]);
                        exit(2);
                }
        }

        printf("DEBUG-Numero processi da creare %d\n", N);

        /* Allocazione dell'array di N pipe descriptors */
        piped = (pipe_t *) malloc (N*sizeof(pipe_t));
        if (piped == NULL)
        {
                printf("Errore nella allocazione della memoria\n");
                exit(3);
        }
```

```c
        /* Creazione delle N pipe figli-padre */
        for (i=0; i < N; i++)
        {
                if (pipe(piped[i]) < 0)
                {
                        printf("Errore nella creazione della pipe\n");
                        exit(4);
                }
        }

        /* Le N pipe nipoti-figli deriveranno dalla creazione di una pipe in ognuno
dei figli che poi creeranno un nipote */

        /* Ciclo di generazione dei figli */
        for (i=0; i < N; i++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork\n");
                        exit(5);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        /* in caso di errori nei figli o nei nipoti decidiamo di
tornare -1 che corrispondera' al valore 255 che non puo' essere un valore
accettabile di ritorno (questo e' garantito dalla parte SHELL) */

                        /* Chiusura delle pipe non usate nella comunicazione con il
padre */
                        for (j=0; j < N; j++)
                        {
                                close(piped[j][0]);
                                if (i != j) close(piped[j][1]);
                        }

                        /* per prima cosa, creiamo la pipe di comunicazione fra
nipote e figlio */
                        if (pipe(p) < 0)
                        {
                                printf("Errore nella creazione della pipe fra
figlio e nipote\n");
                                exit(-1); /* si veda commento precedente */
                        }

                        if ( (pid = fork()) < 0) /* ogni figlio crea un nipote */
                        {
                                printf("Errore nella fork di creazione del
nipote\n");
                                exit(-1); /* si veda commento precedente */
                        }
                        if (pid == 0)
                        {
                                /* codice del nipote */
                                /* in caso di errori nei figli o nei nipoti
decidiamo di tornare -1 che corrispondera' al valore 255 che non puo' essere un
valore accettabile di ritorno */
```

```
                                /* chiusura della pipe rimasta aperta di
comunicazione fra figlio-padre che il nipote non usa */
                                close(piped[i][1]);
                                srand(time(NULL)); /* inizializziamo il seme per la
generazione random di numeri (come indicato nel testo) */
                                /* il nipote ricava dal parametro 'giusto' il
numero di linee del suo file e quindi calcola in modo random il numero di linee che
inviera' al figlio */
                                X=atoi(argv[(i*2)+2]);
                                r=mia_random(X);
                                /* Costruiamo la stringa di opzione per il comando
head */
                                sprintf(opzione, "-%d", r);
                                /* ogni nipote deve simulare il piping dei comandi
nei confronti del figlio e quindi deve chiudere lo standard output e quindi usare
la dup sul lato di scrittura della propria pipe */
                                close(1);
                                dup(p[1]);
                                /* ogni nipote adesso puo' chiudere entrambi i lati
della pipe: il lato 0 NON viene usato e il lato 1 viene usato tramite lo standard
ouput */
                                close(p[0]);
                                close(p[1]);

                                /* si puo' valutare se procedere con la ridirezione
dello standard error su /dev/null: nel caso le istruzioni da aggiungere sono
                                   close(2);
                                   open("/dev/null", O_WRONLY); */

                                /* Il nipote diventa il comando head: bisogna usare
le versioni dell'exec con la p in fondo in modo da usare la variabile di ambiente
PATH */                                                 /* le stringhe che
rappresentano i nomi dei file si trovano con l'indice i moltiplicato per 2 e
sommando 1, cioe' nelle posizioni 1, 3, 5, etc. e quindi saltando le stringhe che
rappresentano i numeri*/
                                execlp("head", "head", opzione, argv[(i*2)+1],
(char *)0);
                                /* attenzione ai parametri nella esecuzione di
head: opzione, nome del file e terminatore della lista */
                                /* NOTA BENE: dato che il testo richiedeva l'uso
del comando head, se per caso il file NON esiste o non fosse leggibile, il comando
head fallirebbe e non riuscirebbe a mandare alcuna linea al figlio e quindi il
valore 0 tornato dal figlio dovrebbe essere interpretato come problemi appunto nel
nipote! */

                                /* Non si dovrebbe mai tornare qui!!*/
                                /* usiamo perror che scrive su standard error, dato
che lo standard output e' collegato alla pipe */
                                perror("Problemi di esecuzione della head da parte
del nipote");
                                exit(-1); /* si veda commento precedente */
                        }
                        /* codice figlio */
                        /* ogni figlio deve chiudere il lato che non usa della pipe
di comunicazione con il nipote */
                        close(p[1]);
                        /* adesso il figlio legge dalla pipe fino a che ci sono
caratteri e cioe' linee inviate dal nipote tramite la head */
                        j=0; /* inizializziamo l'indice della linea */
```

```
                        ritorno=0; /* inizializziamo il numero di linee lette che
alla fine rappresentera' il numero random calcolato dal nipote */
                        S.pid_nipote=pid; /* inizializziamo il campo con il pid del
processo nipote */
                        while (read(p[0], &(S.linea_letta[j]), 1))
                        {
                                /* se siamo arrivati alla fine di una linea */
                                if (S.linea_letta[j] == '\n')
                                {
                                        /* si deve incrementare il valore di
ritorno che rappresenta anche il numero di linee */
                                        ritorno++;
                                        /* nell'array linea_letta abbiamo l'ultima
linea ricevuta: incrementiamo la sua lunghezza per inserire il terminatore di
stringa. Infatti ... */
                                        j++;
                                        S.linea_letta[j]='\0'; /* poiche' il padre
deve stampare la linea su standard output, decidiamo che il figlio converta tale
linea in una stringa per facilitare il compito del padre */
                                        S.numero_linea=ritorno;
                                        /* il figlio comunica al padre */
                                        write(piped[i][1], &S, sizeof(S));
                                        j=0; /* riazzeriamo l'indice per la
prossima linea */
                                }
                                else
                                        j++; /* incrementiamo l'indice della linea
*/
                        }

                        /* il figlio deve aspettare il nipote per correttezza */
                        /* se il nipote e' terminato in modo anomalo decidiamo di
tornare -1 che verra' interpretato come 255 e quindi segnalando questo problema al
padre */
                        if ((pid = wait(&status)) < 0)
                        {
                                printf("Errore in wait\n");
                                ritorno=-1; /* si veda commento precedente */
                        }
                        if ((status & 0xFF) != 0)
                        {
                                printf("Nipote con pid %d terminato in modo
anomalo\n", pid);
                                ritorno=-1; /* si veda commento precedente */
                        }
                        else
                                printf("DEBUG-Il nipote con pid=%d ha ritornato
%d\n", pid, (int)((status >> 8) & 0xFF));
                        exit(ritorno); /* il figlio ritorna il numero di linee
ricevute dal nipote che rappresentano implicitamente il numero random calcolato dal
nipote, come richiesto dal testo */
                }
        }

        /* Codice del padre */
        /* Il padre chiude i lati delle pipe che non usa */
        for (i=0; i < N; i++)
                close(piped[i][1]);
```

```
        /* Il padre recupera le informazioni dai figli: prima in ordine di
strutture e quindi in ordine di indice */
        finito = 0; /* all'inizio supponiamo che non sia finito nessun figlio */
        while (!finito)
        {
                finito = 1; /* mettiamo finito a 1 perche' potrebbero essere finiti
tutti i figli */
                for (i=0; i<N; i++)
                {
                 /* si legge la struttura inviata  dal figlio i-esimo */
                        nr = read(piped[i][0], &S, sizeof(S));
                        /* si deve controllare se il padre ha letto qualcosa, dato
che se un figlio e' terminato non inviera' piu' nulla e quindi il padre NON puo'
eseguire la stampa */
                        if (nr != 0)
                        {
                                finito = 0; /* almeno un processo figlio non e'
finito */
                                /* Nota bene: la stampa della linea con il formato
%s NON ha problemi perche' il figlio ha passato una stringa */
                                printf("Il nipote con pid %d ha letto dal file %s
nella riga %d questa linea:\n%s",  S.pid_nipote, argv[(i*2)+1], S.numero_linea,
S.linea_letta); /* NOTA BENE: dato che la linea contiene il terminatore di linea
nella printf NON abbiamo inserito lo \n alla fine */
                        }
                }
         }

        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                if ((pid = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(6);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {
                        ritorno=(int)((status >> 8) &   0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi nel figlio o nel nipote; se 0 problemi nel nipote)\n", pid, ritorno);
                }
        }
        exit(0);
}
```

# 26 MAGGIO 2017

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 2, da controllare) che rappresentano i primi N nomi di file (F1, F2, ... FN) mentre l'ultimo rappresenta un singolo carattere (Cx) (da controllare). Il processo padre deve generare N processi figli (P0, P1, … PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file Fk (con

k= i+1). Ogni processo figlio Pi deve leggere i caratteri del file associato Fk calcolando (in termini di long int) le occorrenze del carattere Cx. Quindi, i processi figli e il processo padre devono attenersi a questo schema di comunicazione a pipeline: il figlio P0 comunica con il figlio P1 che comunica con il figlio P2 etc. fino al figlio PN-1 che comunica con il padre. Questo schema a pipeline deve prevedere l'invio in avanti di una singola struttura dati, che deve contenere tre campi: 1) c1, di tipo long int, che deve contenere il valore massimo di occorrenze calcolato dal processo Pi; 2) c2, di tipo int, che deve contenere l'indice d'ordine (i) del processo che ha calcolato il massimo; 3) c3, di tipo long int, che deve contenere la somma di tutte le occorrenze calcolate dai processi. Quindi la comunicazione deve avvenire in particolare in questo modo: il figlio P0, dopo aver calcolato il numero di occorrenze occ0 del carattere Cx trovate nel file F1, passa in avanti (cioè comunica) (con una singola write) al processo P1 una struttura S0, con c1 uguale a occ0, c2 uguale a 0 e c3 uguale occ0; il figlio seguente P1, dopo aver calcolato il numero di occorrenze occ1 del carattere Cx trovate nel file F2, deve leggere (con una singola read) la struttura S0 inviata da P0 e quindi deve confezionare la struttura S1 con c1 uguale al massimo fra occ0 e occ1, c2 uguale all'indice del processo che ha calcolato il valore di c1 e c3 uguale alla somma di occ0 e occ1 e la passa (con una singola write) al figlio seguente P2, etc. fino al figlio PN-1, che si comporta in modo analogo, ma passa al padre. Quindi, al processo padre deve arrivare la struttura SN-1. Il padre deve riportare i dati di tale struttura su standard output insieme al pid del processo che ha calcolato il massimo numero di occorrenze e al nome del file in cui sono state trovate (inserendo opportune spiegazioni per l'utente). Al termine, ogni processo figlio Pi deve ritornare al padre il valore intero corrisponde al proprio indice d'ordine (i); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tag**: pipeline dal primo figlio all'ultimo e poi al padre (una singola struttura).

```c
/* Soluzione della parte C del compito della II prova in itinere del 26 Maggio 2017
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>

typedef int pipe_t[2];
typedef struct {
        long int occ;           /* numero massimo di occorrenze (campo c1 del
testo) */
        int id;                 /* indice figlio (campo c2 del testo) */
        long int somma;         /* somma del numero di occorrenze (campo c3 del
testo) */
                } s_occ;

int main (int argc, char **argv)
{
   int N;                               /* numero di file */
   int *pid;                            /* array di pid per fork: N.B. in questo
caso serve un array di pid perche' il padre ricevuta la struttura dalla pipe deve
riportare il PID del processo che ha calcolato il massimo! */
   pipe_t *pipes;                       /* array di pipe usate a pipeline da primo
figlio, a secondo figlio .... ultimo figlio e poi a padre: ogni processo (a parte
il primo) legge dalla pipe i-1 e scrive sulla pipe i */
   int i,j;                             /* indici */
   int fd;                              /* file descriptor */
   int pidFiglio, status, ritorno;      /* per valore di ritorno figli */
   char Cx, ch;                         /* carattere da cercare e carattere letto
da file */
   long int cur_occ;                    /* conteggio delle occorrenze calcolate da
```

```
ogni figlio */
   s_occ letta;                         /* struttura usata dai figli e dal padre */
   int nr,nw;                           /* variabili per salvare valori di ritorno
di read/write da/su pipe */

        /* controllo sul numero di parametri almeno 2 file e un carattere */
        if (argc < 4)
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno tre
parametri (due file e un carattere)\n", argc);
                exit(1);
        }

        /* controlliamo che l'ultimo parametro sia un singolo carattere */
        if (strlen(argv[argc-1]) != 1)
        {
                printf("Errore ultimo parametro non singolo carattere: %s\n",
argv[argc-1]);
                exit(2);
        }

        /* individuiamo il carattere da cercare */
        Cx = argv[argc-1][0];
        printf("DEBUG-Carattere da cercare %c\n", Cx);

        N = argc-2;     /* calcoliamo il numero di file e quindi il numero di
processi figli da creare */
        printf("DEBUG-Numero di processi da creare %d\n", N);

        /* allocazione pid: il padre deve salvare i pid dei figli perche' deve
risalire al pid di ogni figlio dal suo indice */
        if ((pid=(int *)malloc(N*sizeof(int))) == NULL)
        {
                printf("Errore allocazione pid\n");
                exit(3);
        }

        /* allocazione pipe */
        if ((pipes=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe\n");
                exit(4);
        }

        /* creazione pipe */
        for (i=0;i<N;i++)
                if(pipe(pipes[i])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(5);
                }

        /* creazione figli: il pid tornato dalla fork viene salvato direttamente
nell'elemento corrente dell'array dinamico di pid; N.B. per i figli questo array
conterra', per ogni elemento, il valore 0! */
        for (i=0;i<N;i++)
        {
                if ((pid[i]=fork())<0)
                {
```

```
                        printf("Errore creazione figlio\n");
                        exit(6);
                }
                if (pid[i]==0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d e sono associato al file
%s\n", getpid(), argv[i+1]);
                        /* nel caso di errore in un figlio decidiamo di ritornare
un valore via via crescente rispetto al massimo valore di i (che e' N-1),
supponendo comunque che sia minore o uguale di 255 */
                        /* chiusura pipes inutilizzate */
                        for (j=0;j<N;j++)
                        {
                                if (j!=i)
                                        close (pipes[j][1]);
                                if ((i == 0) || (j != i-1))
                                        close (pipes[j][0]);
                        }

                        /* inizializziamo il contatore di occorrenze */
                        cur_occ= 0L;

                        /* apertura file */
                        if ((fd=open(argv[i+1],O_RDONLY))<0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[i+1]);
                                exit(N);
                        }

                        /* ogni figlio deve leggere il proprio file */
                        while(read(fd,&ch,1))
                        {
                                /* cerco il carattere */
                                if (ch == Cx)
                                {
                                        cur_occ++;      /* trovato e quindi si
incrementa il conteggio */
                                }
                        }
                        if (i == 0)
                        {       /* il figlio di indice 0 deve preparare la
struttura da mandare al figlio seguente: usiamo letta dato che comunque il primo
figlio la usa solo per scrivere al figlio successsivo */
                                letta.id = 0;
                                letta.occ = cur_occ;
                                letta.somma = cur_occ;
                        }
                        else
                        {
                                /* lettura da pipe della struttura per tutti i
figli, a parte il primo */
                                nr=read(pipes[i-1][0],&letta,sizeof(s_occ));
                                /* N.B. In caso di pipeline e' particolarmente
importante che la lettura sia andata a buon fine e che quindi la pipeline non si
sia 'rotta' */
                                if (nr != sizeof(s_occ))
                                {
                                        printf("Figlio %d ha letto un numero di
```

```
byte sbagliati %d\n", i, nr);
                                        exit(N+1);
                                }
                                if (letta.occ < cur_occ)
                                {       /* il figlio di indice i ha calcolato un
numero di occorrenze maggiore e quindi bisogna aggiornare i valori di letta */
                                        letta.id = i;
                                        letta.occ = cur_occ;
                                }
                                /* il valore della somma, va aggiornato comunque */
                                letta.somma += cur_occ;
                        }
                        /* tutti i figli mandano in avanti (cioe' al figlio
successivo, a partte l'ultimo figlio che manda al padre) una singola struttura */
                        nw=write(pipes[i][1],&letta,sizeof(s_occ));
                        /* N.B. In caso di pipeline e' particolarmente importante
che anche la scrittura sia andata a buon fine e che quindi la pipeline non si sia
'rotta' */
                        if (nw != sizeof(s_occ))
                        {
                                printf("Figlio %d ha scritto un numero di byte
sbagliati %d\n", i, nw);
                                exit(N+2);
                        }
                        /* ogni figlio deve ritornare al padre il proprio indice
d'ordine */
                        exit(i);
                }
        } /* fine for */

        /* codice del padre */
        /* chiusura pipe: tutte meno l'ultima in lettura */
        for(i=0;i<N;i++)
        {
                close (pipes[i][1]);
                if (i != N-1) close (pipes[i][0]);
        }

        /* il padre deve leggere la singola struttura che gli arriva dall'ultimo
figlio */
        nr=read(pipes[N-1][0],&letta,sizeof(s_occ));
        /* N.B. anche il padre deve controllare di avere letto qualcosa! */
        if (nr != sizeof(s_occ))
        {
                printf("Padre ha letto un numero di byte sbagliati %d\n", nr);
                /* deciso di commentare questa
                exit(7);
                cosi' se i figli incorrono in un qualche problema e quindi la
pipeline si rompe, il padre comunque eseguira' l'attesa dei figli */

        }
        else
        {
                /* il padre deve stampare i campi della struttura ricevuta */
                printf("Il figlio di indice %d e pid %d ha trovato il numero
massimo di occorrenze %ld del carattere '%c' nel file %s\n", letta.id,
pid[letta.id], letta.occ, Cx, argv[letta.id+1]);
                printf("I figli hanno trovato in totale %ld occorrenze del
carattere '%c' nei file\n", letta.somma, Cx);
```

```
                /* il padre poteva anche fare una sola printf dove inserire tutte
le informazioni */
        }

        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                if ((pidFiglio = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(8);
                }
                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se > di %d
problemi)\n", pidFiglio, ritorno, N-1);
                }
        }

        exit(0);
}
```

# 12 FEBBRAIO 2016

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 2, da controllare) che rappresentano N nomi di file (F1, F2. ... FN), mentre l'ultimo rappresenta un carattere alfabetico minuscolo (Cx) (da controllare). Il processo padre deve generare N processi figli (P0, P1, … PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file Fj (con j= i+1). Ogni processo figlio Pi deve leggere i caratteri del file associato Fj cercando il carattere Cx. I processi figli e il processo padre devono attenersi a questo schema di comunicazione a pipeline: il figlio P0 comunica con il figlio P1 che comunica con il figlio P2 etc. fino al figlio PN-1 che comunica con il padre. Questo schema a pipeline deve prevedere l'invio in avanti di un array di strutture dati ognuna delle quali deve contenere due campi: 1) c1, di tipo int, che deve contenere l'indice d'ordine dei processi; 2) c2, di tipo long int, che deve contenere il numero di occorrenze del carattere Cx calcolate dal corrispondente processo. Gli array di strutture DEVONO essere creati da ogni figlio della dimensione minima necessaria per la comunicazione sia in ricezione che in spedizione. Quindi la comunicazione deve avvenire in particolare in questo modo: il figlio P0 passa in avanti (cioè comunica) un array di strutture A1, che contiene una sola struttura con c1 uguale a 0 e con c2 uguale al numero di occorrenze del carattere Cx trovate da P0 nel file F1; il figlio seguente P1, dopo aver calcolato numero di occorrenze del carattere Cx nel file F2, deve leggere (con una singola read) l'array A1 inviato da P0 e quindi deve confezionare l'array A2 che corrisponde all'array A1 aggiungendo all'ultimo posto la struttura con i propri dati e la passa (con una singola write) al figlio seguente P2, etc. fino al figlio PN-1, che si comporta in modo analogo, ma passa al padre. Quindi, al processo padre deve arrivare l'array AN di N strutture (uno per ogni processo P0 … PN-1). Il padre deve riportare i dati di ognuna delle N strutture su standard output insieme al pid del processo corrispondente e al carattere Cx. Al termine, ogni processo figlio Pi deve ritornare al padre il valore intero corrisponde al proprio indice d'ordine (i); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tag**: pipeline dal primo figlio all'ultimo e poi al padre (un array di strutture).

```
/* Soluzione della parte C del compito del 12 Febbraio 2016 */
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>

typedef int pipe_t[2];
typedef struct {
        int id;         /* indice figlio (campo c1 del testo) */
        long int occ;   /* numero occorrenze (campo c2 del testo) */
                } s_occ;

int main (int argc, char **argv)
{
   int N;          /* numero di file */
   int *pid;       /* array di pid per fork */
                   /* l'array di pid serve perche' il padre deve ricavare il pid
dall'indice del array che ricevera' dall'ultimo figlio */
   pipe_t *pipes;  /* array di pipe usate a pipeline da primo figlio, a secondo
figlio .... ultimo figlio e poi a padre: ogni processo (a parte il primo) legge
dalla pipe i-1 e scrive sulla pipe i */
   int i,j;        /* indici */
   int fd;         /* file descriptor */
   int pidFiglio, status, ritorno; /* per valore di ritorno figli */
   char Cx, ch;    /* carattere da cercare e carattere letto da linea */
   s_occ *cur;     /* array di strutture usate dai figli e dal padre: ogni processo
crea l'array della dimensione minima per le sue esigenze! */
   int nr, nw;     /* variabile per salvare valori di ritorno di read e write su/da
pipe */

        /* controllo sul numero di parametri almeno 2 file e un carattere */
        if (argc < 4)
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno tre
parametri (due file e un carattere)\n", argc);
                exit(1);
        }

        /* controlliamo che l'ultimo parametro sia un singolo carattere */
        if (strlen(argv[argc-1]) != 1)
        {
                printf("Errore ultimo parametro non singolo carattere: %s\n",
argv[argc-1]);
                exit(2);
        }

        /* individuiamo il carattere da cercare */
        Cx = argv[argc-1][0];
        /* controlliamo che sia un carattere alfabetico minuscolo */
        if (! islower(Cx)) /* N.B. per usare questa funzione bisogna includere
ctype.h */
        {
                printf("Errore ultimo parametro non alfabetico minuscolo\n");
                exit(3);
        }
        printf("DEBUG-Carattere da cercare %c\n", Cx);
```

```c
        N = argc-2;     /* calcoliamo il numero di file e quindi il numero di
processi figli da creare */
        printf("DEBUG-Numero di processi da creare %d\n", N);

        /* allocazione pid: il padre deve salvare i pid dei figli perche' deve
risalire al pid di ogni figlio dal suo indice */
        if ((pid=(int *)malloc(N*sizeof(int))) == NULL)
        {
                printf("Errore allocazione pid\n");
                exit(4);
        }

        /* allocazione pipe */
        if ((pipes=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe\n");
                exit(5);
        }

        /* creazione pipe */
        for (i=0;i<N;i++)
                if(pipe(pipes[i])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(6);
                }

        /* creazione figli: il pid tornato dalla fork viene salvato direttamente
nell'elemento corrente dell'array dinamico di pid; N.B. per i figli questo array
conterra', per ogni elemento, il valore 0! */
        for (i=0;i<N;i++)
        {
                if ((pid[i]=fork())<0)
                {
                        printf("Errore creazione figlio\n");
                        exit(7);
                }
                if (pid[i]==0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d e sono associato al file
%s\n", getpid(), argv[i+1]);
                        /* nel caso di errore in un figlio decidiamo di ritornare
un valore via via crescente rispetto al massimo valore di i (cioe' N-1), supponendo
comunque che sia minore o uguale di 255 */
                        /* chiusura pipes inutilizzate */
                        for (j=0;j<N;j++)
                        {
                                if (j!=i)
                                        close (pipes[j][1]);
                                if ((i == 0) || (j != i-1))
                                        close (pipes[j][0]);
                        }

                        /* creiamo un array di dimensione 'i+1' anche se leggeremo
'i' strutture, dato che poi ci servira' riempire la 'i+1-esima' struttura! */
                        if ((cur=(s_occ *)malloc((i+1)*sizeof(s_occ))) == NULL)
                        {
                                printf("Errore allocazione cur\n");
                                exit(N);
```

```
                        }
                        /* inizializziamo l'ultima struttura che e' quella
specifica del figlio corrente (nel caso del primo figlio sara' l'unica struttura)
*/
                        cur[i].id = i;
                        cur[i].occ= 0L;

                        /* apertura file */
                        if ((fd=open(argv[i+1],O_RDONLY))<0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[i+1]);
                                exit(N+1);
                        }

                        /* ogni figlio deve leggere il proprio file */
                        while(read(fd,&ch,1)>0)
                        {
                                /* cerco il carattere */
                                if (ch == Cx)
                                {
                                        (cur[i].occ)++; /* trovato e quindi si
incrementa il conteggio */
                                }
                        }
                        if (i != 0)
                        /* lettura da pipe dal figlio precedente dell'array di
strutture per tutti i figli a parte il primo */
                        {
                                nr=read(pipes[i-1][0],cur,i*sizeof(s_occ));
                                if (nr != i*sizeof(s_occ))
                                {
                                        printf("Figlio %d ha letto un numero di
strutture sbagliate %d\n", i, nr);
                                        exit(N+2);
                                }
                        }

                        /* tutti i figli mandano in avanti (cioe' al figlio
successivo, a partte l'ultimo figlio che manda al padre) un array di strutture (per
tutti i figli, a parte il primo, sono 'i' strutture ricevute dal processo
precedente e la 'i+1-esima' che e' la propria); nel caso del primo figlio l'array
mandato avanti contiene un solo elemento costituito ddalla propria struttura */
                        nw=write(pipes[i][1],cur,(i+1)*sizeof(s_occ));
                        if (nw != (i+1)*sizeof(s_occ))
                        {
                                printf("Figlio %d ha scritto un numero di strutture
sbagliate %d\n", i, nr);
                                exit(N+3);
                        }
                        exit(i); /* ogni figlio deve ritornare al padre il proprio
indice d'ordine */
                }
        } /* fine for */

        /* codice del padre */
        /* chiusura pipe: tutte meno l'ultima in lettura */
        for(i=0;i<N;i++)
        {
```

```
                close (pipes[i][1]);
                if (i != N-1) close (pipes[i][0]);
        }

        /* allocazione dell'array di strutture specifico per il padre */
        /* creiamo un array di dimensione N quanto il numero di figli! */
        if ((cur=(s_occ *)malloc(N*sizeof(s_occ))) == NULL)
        {
                printf("Errore allocazione cur nel padre\n");
                exit(8);
        }

        /* il padre deve leggere l'array di strutture che gli arriva dall'ultimo
figlio */
        /* quindi al padre arriva una singola informazione, rappresentata pero' da
un array che quindi contiene piu' elementi */
        nr=read(pipes[N-1][0],cur,N*sizeof(s_occ));
        /* N.B. anche il padre deve controllare di avere letto qualcosa! */
        if (nr != N*sizeof(s_occ))
        {
                printf("Padre ha letto un numero di strutture sbagliate %d\n", nr);
                /* commentiamo questa exit(9); in modo che il padre comunque
aspetti i figli */
        }
        else
        {
                nr=nr/sizeof(s_occ);
                printf("DEBUG-Padre ha letto un numero %d di strutture\n", nr);
                /* il padre deve stampare i campi delle strutture ricevute */
                for(i=0;i<N;i++)
                        printf("Il figlio di indice %d e pid %d ha trovato %ld
occorrenze del carattere %c nel file %s\n", cur[i].id, pid[cur[i].id], cur[i].occ,
Cx, argv[(cur[i].id)+1]);
/* OSSERVAZIONE: in questo caso, il valore di cur[i].id corrisponde al valore di i,
ma se per caso il padre avesse dovuto ordinare l'array ricevuto secondo il campo
occ e, poi solo dopo, avesse dovuto effettuare questa stampa, questo non sarebbe
piu' stato vero; in ogni modo, bisogna sempre rispettare la specifica indicata! */
        }

        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                if ((pidFiglio = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(10);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se > di %d
problemi)\n", pidFiglio, ritorno, N-1);
                }
        }
```

```
        exit(0);
}
```

# 13 GIUGNO 2018

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 2, da controllare) che rappresentano N nomi di file (F1, F2. ... FN-1), mentre l'ultimo rappresenta un numero intero strettamente positivo (Y) (da controllare) che indica la lunghezza in linee dei file: infatti, la lunghezza in linee dei file è la stessa (questo viene garantito dalla parte shell e NON deve essere controllato). Il processo padre deve generare N processi figli (P0, P1, … PN-1): i processi figli Pi (con i che varia da 0 a N-

1. sono associati agli Nfile Ff (con f= i+1). Ogni processo figlio Pi deve leggere tutte le Y linee del file associato Ff calcolando la lunghezza di ogni linea, compreso il terminatore di linea. I processi figli e il processo padre devono attenersi a questo schema di comunicazione a pipeline: il figlio P0 comunica con il figlio P1 che comunica con il figlio P2 etc. fino al figlio PN-1 che comunica con il padre. Questo schema a pipeline deve prevedere l'invio in avanti, per ognuna delle Y linee dei file, di un array di strutture dati ognuna delle quali deve contenere due campi: 1) c1, di tipo int, che deve contenere il pid del processo figlio; 2) c2, di tipo int, che deve contenere la lunghezza della linea corrente, compreso il terminatore di linea. Gli array di strutture DEVONO essere creati da ogni figlio della dimensione minima necessaria per la comunicazione sia in ricezione che in spedizione. Quindi la comunicazione deve avvenire in particolare in questo modo: il figlio P0 passa in avanti (cioè comunica), per la prima linea, un array di strutture A1, che contiene una sola struttura con c1 uguale al suo pid e con c2 uguale alla lunghezza della prima linea (compreso il terminatore di linea) del file F1; il figlio seguente P1, dopo aver calcolato la lunghezza della sua prima linea (compreso il terminatore di linea) del file F2, deve leggere (con una singola read) l'array A1 inviato da P0 e quindi deve confezionare l'array A2 che corrisponde all'array A1 aggiungendo all'ultimo posto la struttura con i propri dati e la passa (con una singola write) al figlio seguente P2, etc. fino al figlio PN-1, che si comporta in modo analogo, ma passa al padre, e così via per ognuna delle altre linee. Quindi, al processo padre deve arrivare, per ognuna delle Y linee dei file, un array AN di N strutture (uno per ogni processo P0 … PN-1). Per ogni array AN ricevuto, il padre deve effettuare un ordinamento in senso crescente in base alle lunghezze e quindi deve riportare su standard output i dati così ordinati di ognuna delle N strutture insieme al numero di linea cui si riferisce l'array e ai file cui si riferiscono i dati. Al termine, ogni processo figlio Pi deve ritornare al padre il valore intero corrisponde al proprio indice d'ordine (i); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tag**: pipeline dal primo all'ultimo figlio e poi al padre al quale arrivano N array di struct in numero NOTO.

```c
/* Soluzione della parte C del compito del 13 Giugno 2018 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>

typedef int pipe_t[2];
typedef struct{
        int pid;        /* pid figlio (campo c1 del testo) */
        int lunghezza;  /* lunghezza linea (campo c2 del testo) */
                } s;

typedef struct{
```

```
        int pid;        /* pid figlio */
        int indice;     /* indice del figlio: CAMPO AGGIUNTO DAL PADRE */
        int lunghezza;  /* lunghezza linea */
                } s_padre;

void bubbleSort(s_padre v[],int dim) /* il tipo degli elementi NON e' un semplice
int come riportato sul testo, ma deve essere il tipo s_padre (appositamente
definito) */
{
int i;
int ordinato=0;
s_padre a;  /* variabile di appoggio per fare lo scambio */
        while (dim>1 && ordinato!=1)
        {
                ordinato=1;
                for (i=0;i<dim-1;i++)
                if (v[i].lunghezza > v[i+1].lunghezza) /* chiaramente il confronto
va fatto sul campo lunghezza della struttura */
                {  /* viene effettuato in questo caso un ordinamento in senso
crescente */
                        /* scambio gli elementi */
                        a=v[i];
                        v[i]=v[i+1];
                        v[i+1]=a;
                        ordinato=0;
                }
                dim--;
        }
}/* fine bubblesort */

int main (int argc, char **argv)
{
int N;                  /* numero di file */
int Y;                  /* numero di linee */
int pid;                /* pid per fork */
pipe_t *pipes;          /* array di pipe usate a pipeline da primo figlio, a
secondo figlio .... ultimo figlio e poi a padre: ogni processo (a parte il primo)
legge dalla pipe i-1 e scrive sulla pipe i */
int i,j;                /* contatori */
int fd;                 /* file descriptor */
int pidFiglio, status, ritorno; /* per valore di ritorno figli */
char ch;                /* carattere letto da linea */
s *cur;                 /* array di strutture usate dai figli */
s_padre *cur_padre;     /* array di strutture usate dal padre */
int nr;                 /* variabili per salvare valori di ritorno di read su pipe
*/

/* controllo sul numero di parametri almeno 2 file e un carattere */
if (argc < 4)
{
        printf("Errore numero di parametri\n");
        exit(1);
}

/* calcoliamo valore di Y e lo controlliamo */
Y=atoi(argv[argc-1]);
if (Y < 0)
{
   printf("Errore valore di Y\n");
```

```
   exit(2);
}

/* stampa di debugging */
printf("Valore di Y %d\n", Y);

N = argc-2;
printf("Numero di processi da creare %d\n", N);

/* allocazione pipe */
if ((pipes=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
{
        printf("Errore allocazione pipe\n");
        exit(3);
}

/* creazione pipe */
for (i=0;i<N;i++)
        if(pipe(pipes[i])<0)
        {
                printf("Errore creazione pipe\n");
                exit(4);
        }

/* creazione figli */
for (i=0;i<N;i++)
{
        if ((pid=fork())<0)
        {
                printf("Errore creazione figli\n");
                exit(5);
        }
        if (pid==0)
        {       /* codice figlio */
                printf("Sono il figlio %d e sono associato al file %s\n", getpid(),
argv[i+1]);
                /* nel caso di errore in un figlio decidiamo di ritornare un valore
via via crescente rispetto al massimo valore di i (cioe' N-1) */
                /* chiusura pipes inutilizzate */
                for (j=0;j<N;j++)
                {
                        if (j!=i)
                                close (pipes[j][1]);
                        if ((i == 0) || (j != i-1))
                                close (pipes[j][0]);
                }

                /* allocazione dell'array di strutture specifico di questo figlio
*/
                /* creiamo un array di dimensione i+1 anche se leggeremo i
strutture, dato che poi ci servira' riempire la i+1-esima struttura! */
                if ((cur=(s *)malloc((i+1)*sizeof(s))) == NULL)
                {
                        printf("Errore allocazione cur\n");
                        exit(N);
                }
                /* inizializziamo l'ultima struttura che e' quella specifica del
figlio corrente (nel caso del primo figlio sara' l'unica struttura */
                cur[i].pid = getpid();
```

```
                cur[i].lunghezza= 0;

                /* apertura file */
                if ((fd=open(argv[i+1],O_RDONLY))<0)
                {
                        printf("Impossibile aprire il file %s\n", argv[i+1]);
                        exit(N+1);
                }
                while(read(fd,&ch,1)>0)
                {
                        /* se leggo un carattere, incremento la lunghezza della
linea */
                        cur[i].lunghezza++;
                        if (ch == '\n')
                        {
                                /* se siamo a fine linea si deve leggere dal figlio
precedente l'array (se non siamo il primo figlio) e mandare la struttura al figlio
successivo */
                                if (i!=0)
                                /* lettura da pipe dell'array di strutture per
tutti i figli a parte il primo */
                                {
                                        nr=read(pipes[i-1][0],cur,i*sizeof(s));
                                        if (nr != i*sizeof(s))
                                        {
                                                printf("Figlio %d ha letto un
numero di strutture sbagliate %d\n", i, nr);
                                                exit(N+2);
                                        }
                                        /*
                                        for(j=0;j<i;j++)
                                                printf("HO ricevuto da figlio di
pid %d la lunghezza %d\n", cur[j].pid, cur[j].lunghezza);
                                        */
                                }

                                /* tutti i figli mandano in avanti, l'ultimo figlio
manda al padre un array di strutture (i ricevute dal processo precedente e la i+1-
esima la propria */
                                write(pipes[i][1],cur,(i+1)*sizeof(s));
                                cur[i].lunghezza= 0;    /* si deve azzerare
nuovamente la lunghezza per la prossima linea */
                        }
                }
                exit(i); /* ogni figlio deve ritornare al padre il proprio indice
*/
        }
} /* fine for */

/* codice del padre */
/* chiusura pipe: tutte meno l'ultima in lettura */
for (i=0;i<N;i++)
{
        close (pipes[i][1]);
        if (i != N-1) close (pipes[i][0]);
}

/* allocazione dell'array di strutture specifico per il padre per la read */
/* creiamo un array di dimensione N quanto il numero di figli! */
```

```
if ((cur=(s *)malloc(N*sizeof(s))) == NULL)
{
        printf("Errore allocazione cur nel padre\n");
        exit(6);
}

/* allocazione dell'array di strutture specifico per il padre per l'ordinamento */
/* creiamo un array di dimensione N quanto il numero di figli! */
if ((cur_padre=(s_padre *)malloc(N*sizeof(s_padre))) == NULL)
{
        printf("Errore allocazione curi_padre nel padre\n");
        exit(7);
}

/* il padre deve leggere via via gli array di strutture che gli arrivano
dall'ultimo figlio: uno per ogni linea */
for (j=1;j<=Y;j++)
{
        nr=read(pipes[N-1][0],cur,N*sizeof(s));
        if (nr != N*sizeof(s))
        {
                printf("Padre ha letto un numero di strutture sbagliate %d\n", nr);
                exit(8);
        }
        /* il padre deve copiare l'array ricevuto nel proprio array */
        for (i=0;i<N;i++)
        {
                cur_padre[i].pid = cur[i].pid;
                cur_padre[i].indice = i;
                cur_padre[i].lunghezza = cur[i].lunghezza;
        }
        /* ordiniamo l'array cur_padre */
        bubbleSort(cur_padre,N);
        /* stampiamo le informazioni ordinate secondo la specifica */
        for (i=0;i<N;i++)
                printf("Il figlio con pid %d per il file %s ha calcolato per la
linea %d la lunghezza %d\n", cur_padre[i].pid, argv[cur_padre[i].indice + 1], j,
cur_padre[i].lunghezza);
}

/* Il padre aspetta i figli */
for (i=0; i < N; i++)
{
        pidFiglio = wait(&status);
        if (pidFiglio < 0)
        {
                printf("Errore in wait\n");
                exit(9);
        }

        if ((status & 0xFF) != 0)
                printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
        else
        {
                ritorno=(int)((status >> 8) & 0xFF);
                printf("Il figlio con pid=%d ha ritornato %d\n", pidFiglio,
ritorno);
        }
}
```

```
exit(0);
}
```

# 7 SETTEMBRE 2022

TESTO PARTE C: ATTENZIONE LEGGERE ANCHE LA NOTA SEGUENTE AL TESTO! La parte in C accetta un numero variabile di parametri N maggiore o uguale a 2 che rappresentano nomi di file (F1, …FN), che hanno tutti la stessa lunghezza dispari (da non controllare). Il processo padre deve, per prima cosa, creare nella directory corrente un file fcreato con nome corrispondente al proprio COGNOME (tutto scritto in maiuscolo, in caso di più cognomi se ne usi solo uno, inserendo un opportuno commento). Il processo padre deve generare un numero di processi figli pari a N: ogni processo figlio Pn è associato ad uno dei file F1, …FN (in ordine). Ognuno di tali processi figli Pn esegue concorrentemente e legge tutti i caratteri del proprio file associato operando una opportuna selezione come indicato nel seguito. I processi figli Pn devono usare uno schema di comunicazione a pipeline: il figlio P0 comunica con il figlio P1 che comunica con il figlio P2 etc. fino al figlio PN-1 che comunica con il padre. Questo schema a pipeline deve prevedere l'invio in avanti, per ognuno dei caratteri dispari, di un array di grandezza N e in cui ogni elemento dell'array corrisponde al carattere corrente car di posizione dispari (il primo carattere ha posizione 1) letto dal corrispondente processo figlio Pn. Quindi, il generico processo Pn, dopo aver letto il carattere dispari corrente, deve ricevere dal figlio precedente (a parte il processo P0) l'array di caratteri dispari e, dopo aver inserito il proprio carattere dispari corrente nella posizione giusta dell'array, deve inviare l'array di caratteri dispari al figlio successivo, con PN-1 che manda al padre. Quindi al padre deve arrivare, per ognuno dei caratteri dispari, un array di caratteri dispari, ognuno dei quali deve essere scritto sul file fcreato. Al termine dell'esecuzione, ogni figlio Pn ritorna al padre l'ultimo carattere dispari letto dal proprio file (sicuramente minore di 255); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

NOTA BENE NEL FILE C main.c SI USI OBBLIGATORIAMENTE:

- una variabile di nome N per il numero di processi figli;
- una variabile di nome n per l'indice dei processi figli;
- una variabile fcreato per il file descriptor del file creato;
- una variabile di nome car per il carattere letto correntemente dai figli dal proprio file;
- una variabile di nome cur per l'array da passare fra i vari figli fino al padre.

**tag**: invio in avanti di un array di linee. Al padre ne arrivano n.

```
/* Soluzione della parte C del compito del 7 Settembre 2022 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>
#define PERM 0644

typedef int pipe_t[2];

int main (int argc, char **argv)
{
   int N;                                /* numero di file/processi */
   /* ATTENZIONE NOME N imposto dal testo! */
   int pid;                              /* pid per fork */
   pipe_t *pipes;                        /* array di pipe usate a pipeline da primo
```

```
figlio, a secondo figlio .... ultimo figlio e poi a padre: ogni processo (a parte
il primo) legge dalla pipe i-1 e scrive sulla pipe i */
   int n,j;                             /* indici */
   /* ATTENZIONE NOME n imposto dal testo! */
   int fd, fcreato;                     /* file descriptor */
   /* ATTENZIONE NOME fcreato imposto dal testo! */
   int pidFiglio, status, ritorno;      /* per valore di ritorno figli */
   int nroCar;                          /* variabile per salvare la lunghezza in
caratteri dei file e quindi capire se si e' su un carattere di posizione dispari */
   char car;                            /* variabile per salvare i caratteri letti
da ogni file */
   /* ATTENZIONE car imposto dal testo! */
   char *cur;                           /* array dinamico di caratteri */
   /* ATTENZIONE NOME cur imposto dal testo! */
   int nr,nw;                           /* variabili per salvare valori di ritorno
di read/write da/su pipe */

        /* controllo sul numero di parametri: almeno 2 file */
        if (argc < 3)
        {
                printf("Errore numero di parametri, argc=%d\n", argc);
                exit(1);
        }

        N = argc-1;
        printf("DEBUG-Numero di processi da creare %d\n", N);

        /* allocazione pipe */
        if ((pipes=(pipe_t *)malloc(N*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe\n");
                exit(2);
        }

        /* creazione pipe */
        for (n=0;n<N;n++)
                if(pipe(pipes[n])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(3);
                }

        /* allocazione array di caratteri: la puo' fare il padre per tutti i figli
*/
        if ((cur=(char *)malloc(N*sizeof(char))) == NULL)
        {
                printf("Errore allocazione array di caratteri\n");
                exit(4);
        }

        /* creiamo il file nella directory corrente avente come nome il mio cognome
(TUTTO IN MAIUSCOLO, come specificato nel testo) */
        if ((fcreato=creat("LEONARDI", PERM)) < 0)
        {
                printf("Errore nella creat del file %s\n", "Leonardi");
                exit(5);
        }

        /* creazione figli */
```

```
        for (n=0;n<N;n++)
        {
                if ((pid=fork())<0)
                {
                        printf("Errore creazione figlio\n");
                        exit(6);
                }
                if (pid == 0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d e sono associato al file
%s\n", getpid(), argv[n+1]);
                        /* nel caso di errore in un figlio decidiamo di ritornare
il valore -1 che sara' interpretato dal padre come 255 (valore NON ammissibile) */

                        /* chiusura pipes inutilizzate */
                        for (j=0;j<N;j++)
                        {       /* si veda commento nella definizione dell'array
pipes per comprendere le chiusure */
                                if (j!=n)
                                        close (pipes[j][1]);
                                if ((n == 0) || (j != n-1))
                                        close (pipes[j][0]);
                        }

                        /* apertura file */
                        if ((fd=open(argv[n+1],O_RDONLY))<0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[n+1]);
                                exit(-1);
                        }

                        /* inizializziamo il numero dei caratteri letti */
                        nroCar = 0;
                        /* con un ciclo leggiamo tutti caratteri, come richiede la
specifica */
                        while(read(fd,&car,1) != 0)
                        {
                                nroCar++;       /* incrementiamo il numero dei
caratteri letti */
                                if ((nroCar % 2) == 1) /* siamo su un carattere di
posizione dispari */
                                {
                                        if (n != 0)
                                        {
                                                /* se non siamo il primo figlio,
dobbiamo aspettare l'array dal figlio precedente per mandare avanti */
                                                nr=read(pipes[n-1]
[0],cur,N*sizeof(char));
                                                /* per sicurezza controlliamo il
risultato della lettura da pipe */
                                                if (nr != N*sizeof(char))
                                                {
                                                        printf("Figlio %d ha letto
un numero di byte sbagliati %d\n", n, nr);
                                                        exit(-1);
                                                }
                                        }
                                        /* a questo punto si deve inserire il
```

```
carattere letto nel posto giusto */
                                        cur[n]=car;

                                        /* ora si deve mandare l'array in avanti */
                                        nw=write(pipes[n][1],cur,N*sizeof(char));
                                        /* anche in questo caso controlliamo il
risultato della scrittura */
                                        if (nw != N*sizeof(char))
                                        {
                                                printf("Figlio %d ha scritto un
numero di byte sbagliati %d\n", n, nw);
                                                exit(-1);
                                        }
                                        ritorno=car; /* salviamo il carattere
dispari perche' potrebbe essere quello da ritornare! IN REALTA' CONSIDERANDO LA
SPECIFICA DELLA PARTE SHELL E CIOE' CHE LA LUNGHEZZA DEI FILE E' DISPARI QUESTO
SALVATAGGIO NON SERVIREBBE E BASTEREBBE TORNARE car CHE E' SICURAMENTE, COME ULTIMO
CARATTERE, IN POSIZIONE DISPARI! */
                                }
                                else
                                ; /* NON SI DEVE FARE NULLA */
                        }
                        /* ogni figlio deve tornare l'ultimo carattere letto */
                        exit(ritorno);
                }
        } /* fine for */

        /* codice del padre */
        /* chiusura di tutte le pipe che non usa */
        for (n=0;n<N;n++)
        {
                close (pipes[n][1]);
                if (n != N-1) close (pipes[n][0]);
        }

        /* il padre deve leggere tutti gli array di caratteri inviati dall'ultimo
figlio */
        while (read(pipes[n-1][0],cur,N*sizeof(char)))
        {
                /* il padre deve scrivere i caratteri sul file creato: si puo'
usare una singola write di N caratteri! */
                write(fcreato, cur, N*sizeof(char));
        }

        /* Il padre aspetta i figli */
        for (n=0; n < N; n++)
        {
                pidFiglio = wait(&status);
                if (pidFiglio < 0)
                {
                        printf("Errore in wait\n");
                        exit(9);
                }
                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
```

```
                    printf("Il figlio con pid=%d ha ritornato %c (e in decimale
%d se 255 problemi)\n", pidFiglio, ritorno, ritorno);
                }
        }
        exit(0);
}
```

# 5 GIUGNO 2015

La parte in C accetta un numero variabile di parametri (maggiore o uguale a 2, da controllare) che rappresentano M nomi assoluti di file F1...FM. Il processo padre deve generare M processi figli (P0 … PM-1): ogni processo figlio è associato al corrispondente file Fi (con i = j+1). Ognuno di tali processi figli deve creare a sua volta un processo nipote (PP0 … PPM-1): ogni processo nipote PPj esegue concorrentemente e deve, usando in modo opportuno il comando tail di UNIX/Linux, leggere l'ultima linea del file associato Fi. Ogni processo figlio Pj deve calcolare la lunghezza, in termini di valore intero (lunghezza), della linea scritta (escluso il terminatore di linea) sullo standard output dal comando tail eseguito dal processo nipote PPj; quindi ogni figlio Pj deve comunicare tale lunghezza al padre. Il padre ha il compito di ricevere, rispettando l'ordine inverso dei file, il valore lunghezza inviato da ogni figlio Pj che deve essere riportato sullo standard output insieme all'indice del processo figlio e al nome del file cui tale lunghezza si riferisce. Al termine, ogni processo figlio Pj deve ritornare al padre il valore di ritorno del proprio processo nipote PPj e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tag**: ogni figlio riceve dal nipote una serie di informazioni (calcolate mediante comandi UNIX) e li manda al padre UNA INFORMAZIONE.

```
/* Soluzione della Prova d'esame del 5 Giugno 2015 - SOLO Parte C */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

typedef int pipe_t[2]; /* definizione del TIPO pipe_t come array di 2 interi */

int main(int argc, char **argv)
{
   /* -------- Variabili locali ---------- */
   int pid;                    /* process identifier per le fork() */
   int M;                      /* numero di file passati sulla riga di comando,
che corrispondera' al numero di figli */
   pipe_t *piped;              /* array dinamico di pipe descriptors per
comunicazioni figli-padre  */
   pipe_t p;                   /* una sola pipe per ogni coppia figlio-nipote */
   int j, k;                   /* indici per i cicli */
   char ch;                    /* carattere per leggere dalla pipe collegata allo
standard output del nipote (tail) e calcolare la lunghezza della ultima linea */
   int l;                      /* variabile che serve per contare caratteri letti
dal figlio (inviati al nipote) */
   int lunghezza;              /* variabile che viene comunicata da ogni figlio al
padre e che contiene la lunghezza dell'ultima linea */
   int status;                 /* variabile di stato per la wait */
   int ritorno;                /* variabile che viene ritornata da ogni figlio al
```

```
padre e che contiene il ritorno del nipote */
   /* ---------------------------------- */

        /* Controllo sul numero di parametri */
        if (argc < 3) /* Meno di due parametri */
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno due
parametri\n", argc);
                exit(1);
        }

        /* Calcoliamo il numero di file passati e quindi di figli da creare */
        M = argc - 1;

        /* Allocazione dell'array di M pipe descriptors */
        piped = (pipe_t *) malloc (M*sizeof(pipe_t));
        if (piped == NULL)
        {
                printf("Errore nella allocazione della memoria\n");
                exit(2);
        }

        /* Creazione delle M pipe figli-padre */
        for (j=0; j < M; j++)
        {
                if(pipe(piped[j]) < 0)
                {
                        printf("Errore nella creazione della pipe\n");
                        exit(3);
                }
        }

        /* Le M pipe nipoti-figli deriveranno dalla creazione di una pipe in ognuno
dei figli che poi creeranno un nipote */

        printf("DEBUG-Sono il processo padre con pid %d e sto per generare %d
figli\n", getpid(), M);

        /* Ciclo di generazione dei figli */
        for (j=0; j < M; j++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork\n");
                        exit(4);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        printf("DEBUG-Sono il processo figlio di indice %d e pid %d
sto per creare il nipote che recuperera' l'ultima linea del file %s\n", j,
getpid(), argv[j+1]);
                        /* in caso di errori nei figli o nei nipoti decidiamo di
tornare dei numeri negativi (-1 che corrispondera' per il padre al valore 255, -2
che corrispondera' a 254, etc.) che non possono essere valori accettabili di
ritorno per il comando tail */

                        /* Chiusura delle pipe non usate nella comunicazione con il
```

```
padre */
                        for (k=0; k < M; k++)
                        {
                                close(piped[k][0]);
                                if (k != j) close(piped[k][1]);
                        }

                        /* per prima cosa, creiamo la pipe di comunicazione fra
nipote e figlio */
                        if (pipe(p) < 0)
                        {
                                printf("Errore nella creazione della pipe fra
figlio e nipote!\n");
                                exit(-1); /* si veda commento precedente */
                        }

                        if ( (pid = fork()) < 0) /* ogni figlio crea un nipote */
                        {
                                printf("Errore nella fork di creazione del
nipote\n");
                                exit(-2); /* si veda commento precedente */
                        }
                        if (pid == 0)
                        {
                                /* codice del nipote */
                                /* in caso di errori nei figli o nei nipoti
decidiamo di tornare dei numeri negativi (-1 che corrispondera' per il padre al
valore 255, -2 che corrispondera' a 254, etc.) che non possono essere valori
accettabili di ritorno per il comando tail */
                                printf("DEBUG-Sono il processo nipote del figlio di
indice %d e pid %d e sto per recuperare l'ultima linea del file %s\n", j, getpid(),
argv[j+1]);
                                /* chiusura della pipe rimasta aperta di
comunicazione fra figlio-padre che il nipote non usa */
                                close(piped[j][1]);
                                /* ogni nipote deve simulare il piping dei comandi
nei confronti del figlio e quindi deve chiudere lo standard output e quindi usare
la dup sul lato di scrittura della propria pipe */
                                close(1);
                                dup(p[1]);
                                /* ogni nipote adesso puo' chiudere entrambi i lati
della pipe: il lato 0 non viene usato e il lato 1 viene usato tramite lo standard
output */
                                close(p[0]);
                                close(p[1]);

                                /* si puo' valutare se procedere con la ridirezione
dello standard error su /dev/null: nel caso le istruzioni da aggiungere sono
                                   close(2);
                                   open("/dev/null", O_WRONLY); */

                                /* Il nipote diventa il comando tail: bisogna usare
le versioni dell'exec con la p in fondo in modo da usare la variabile di ambiente
PATH */
                                execlp("tail", "tail", "-1", argv[j+1], (char *)0);
                                /* attenzione ai parametri nella esecuzione di
tail: -1, come opzione, il nome del file e terminatore della lista */

                                /* Non si dovrebbe mai tornare qui!! */
```

```
                                /* usiamo perror che scrive su standard error, dato
che lo standard output e' collegato alla pipe */
                                perror("Problemi di esecuzione della head da parte
del nipote");
                                exit(-3); /* si veda commento precedente */
                        }
                        /* codice figlio */
                        /* ogni figlio deve chiudere il lato che non usa della pipe
di comunicazione con il nipote */
                        close(p[1]);
                        /* adesso il figlio legge dalla pipe fino a che ci sono
caratteri inviati dal nipote tramite la tail -1 */
                        l=0;
                        while (read(p[0], &ch, 1))
                        {
                                l++;
                        }
                        if (l!=0) /* se il figlio ha letto qualcosa */
                        {
                                lunghezza=l-1;  /* decrementando di 1 il valore di
l otteniamo la lunghezza della linea escluso il terminatore; se teniamo invece l
avremmo la lunghezza della linea compreso il terminatore */
                        }
                        else
                        {
                                lunghezza=0;
                        }

                        /* il figlio comunica al padre */
                        write(piped[j][1], &lunghezza, sizeof(lunghezza));

                        /* il figlio deve aspettare il nipote per restituire il
valore al padre */
                        /* se il nipote e' terminato in modo anomalo decidiamo di
tornare -1 che verra' interpretato come 255 e quindi segnalando questo problema al
padre */
                        ritorno=-1;
                        if ((pid = wait(&status)) < 0)
                        {
                                printf("Errore in wait\n");
                                exit(-5);
                        }
                        if ((status & 0xFF) != 0)
                                printf("Nipote con pid %d terminato in modo
anomalo\n", pid);
                        else
                                printf("DEBUG-Il nipote con pid=%d ha ritornato
%d\n", pid, ritorno=(int)((status >> 8) & 0xFF));
                        exit(ritorno);
                }
        }

        /* Codice del padre */
        /* Il padre chiude i lati delle pipe che non usa */
        for (j=0; j < M; j++)
                close(piped[j][1]);

        /* Il padre recupera le informazioni dai figli in ordine inverso di indice
*/
```

```
        for (j=M-1; j >= 0; j--)
        {
                /* il padre recupera tutti i valori interi dai figli */
                read(piped[j][0], &lunghezza, sizeof(lunghezza));
                printf("Il figlio di indice %d ha comunicato il valore %d per il
file %s\n", j, lunghezza, argv[j+1]);
        }

        /* Il padre aspetta i figli */
        for (j=0; j < M; j++)
        {
                if ((pid = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {
                        ritorno=(int)((status >> 8) &   0xFF);
                        if (ritorno!=0)
                                printf("Il figlio con pid=%d ha ritornato %d e
quindi vuole dire che il nipote non e' riuscito ad eseguire il tail oppure il
figlio o il nipote sono incorsi in errori\n", pid, ritorno);
                        else    printf("Il figlio con pid=%d ha ritornato %d\n",
pid, ritorno);
                }
        }
        exit(0);
}
```

# 12 SETTEMBRE 2018

La parte in C accetta un numero variabile N di parametri (con N maggiore o uguale a 2, da controllare) che rappresentano N nomi di file (F1, F2. ... FN). Il processo padre deve generare N processi figli Pi (P0 … PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file Ff (con f= i+1). Ognuno di tali processi figli deve creare a sua volta un processo nipote PPi (PP0 … PPN-1) associato sempre al corrispondente file Ff. Ogni processo figlio Pi e ogni nipote PPi esegue concorrentemente andando a cercare nel file associato Ff tutte le occorrenze dei caratteri numerici per il figlio e tutte le occorrenze dei caratteri alfabetici minuscoli per il nipote. Ognuno dei processi figlio e nipote deve operare una modifica del file Ff: in specifico, ogni nipote deve trasformare ogni carattere alfabetico minuscolo nel corrispondente carattere alfabetico maiuscolo, mentre ogni figlio deve trasformare ogni carattere numerico nel carattere spazio. Una volta terminate le trasformazioni, sia i processi figli Pi che i processi nipoti PPi devono comunicare al padre il numero (in termini di long int) di trasformazioni effettuate. Il padre ha il compito di stampare su standard output, rispettando l'ordine dei file, il numero di trasformazioni ricevute da ogni figlio Pi e da ogni nipote PPi, riportando opportuni commenti esplicativi, che devono includere anche il nome del file che è stato interessato dalle trasformazioni. Al termine, ogni processo nipote PPi deve ritornare al figlio Pi un opportuno codice ed analogamente ogni processo figlio Pi deve ritornare al padre un opportuno codice; il codice che ogni nipote PPi e ogni figlio Pi deve ritornare è: a) 0 se il numero di trasformazioni attuate è minore di 256; b) 1 se il numero di trasformazioni attuate è maggiore o uguale a 256, ma minore di 512; c) 2 se il numero di trasformazioni attuate è maggiore o uguale a 512, ma minore di 768; d) etc. Sia ogni figlio Pi e sia il padre devono stampare su standard output il PID di ogni nipote/figlio e il valore ritornato.

**tag**: il padre riceve da ogni figlio una singola informazione, il padre riceve da ogni nipote una singola informazione.

```c
/* Soluzione della Prova d'esame del 12 Settembre 2018 - Parte C */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <ctype.h>

typedef int pipe_t[2];

int main(int argc, char **argv)
{
   /* -------- Variabili locali ---------- */
   int pid;                     /* process identifier per le fork() */
   int N;                       /* numero di file passati sulla riga di comando e
quindi di figli da creare */
   pipe_t *pipedFP;             /* array dinamico di pipe descriptors per
comunicazioni figli-padre  */
   pipe_t *pipedNP;             /* array dinamico di pipe descriptors per
comunicazioni nipoti-padre  */
   int fd;                      /* file descriptor che serve sia ai figli che ai
nipoti */
   int i, j;                    /* indici per i cicli */
   char ch;                     /* variabile che serve per leggere i caratteri sia
da parte dei figli che dei nipoti */
   long int trasformazioni=0L;  /* variabile che mantiene il numero delle
trasformazioni effettuate sia dai figli che dai nipoti */
   int ritorno;                 /* variabile che viene ritornata da ogni figlio al
padre e da ogni nipote al figlio */
   int status;                  /* variabile di stato per la wait */
   /* ------------------------------------ */

        /* Controllo sul numero di parametri (ci devono essere almeno di due
parametri */
        if (argc < 3) /* Meno di due parametri */
        {
                printf("Errore nel numero dei parametri, dato che argc=%d (ci
vogliono almeno due nomi di file)\n", argc);
                exit(1);
        }

        /* Calcoliamo il numero di file passati e quindi di figli da creare */
        N = argc - 1;

        /* Allocazione dell'array di N pipe descriptors figli-padre */
        pipedFP = (pipe_t *) malloc(N*sizeof(pipe_t));
        if (pipedFP == NULL)
        {
                printf("Errore nella allocazione della memoria pipe figli-
padre\n");
                exit(2);
        }
```

```
        /* Allocazione dell'array di N pipe descriptors nipoti-padre */
        /* N.B. In questo caso i nipoti devono comunicare direttamente con il padre
e quindi e' il padre che deve creare una pipe per ogni nipote e non il figlio come
invece si puo' fare nei testi in cui la comunicazione deve essere nipote-figlio */
        pipedNP = (pipe_t *) malloc (N*sizeof(pipe_t));
        if (pipedNP == NULL)
        {
                printf("Errore nella allocazione della memoria pipe nipoti-
padre\n");
                exit(3);
        }

        /* Creazione delle N pipe figli-padre e delle N pipe nipoti-padre */
        for (i=0; i < N; i++)
        {
                if (pipe(pipedFP[i]) < 0)
                {
                        printf("Errore nella creazione della pipe %d-esima figli-
padre\n", i);
                        exit(4);
                }
                if (pipe(pipedNP[i]) < 0)
                {
                        printf("Errore nella creazione della pipe %d-esima nipoti-
padre\n", i);
                        exit(5);
                }
        }

        /* Ciclo di generazione dei figli */
        for (i=0; i < N; i++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork %d-esima\n", i);
                        exit(6);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        printf("DEBUG-Sono il processo figlio di indice %d e pid %d
sto per creare il nipote che leggera' sempre dal mio stesso file %s\n", i,
getpid(), argv[i+1]);
                        /* Chiusura delle pipe non usate nella comunicazione con il
padre  */
                        for (j=0; j < N; j++)
                        {
                                close(pipedFP[j][0]);
                                if (i != j) close(pipedFP[j][1]);
                        }

                        /* N.B. Le chiusure delle altre pipe vanno fatte
assolutamente dopo la creazione del nipote! */
                        if ( (pid = fork()) < 0)
                        {
                                printf("Errore nella fork di creazione del
nipote\n");
                                exit(-1); /* decidiamo, in caso di errore, di
```

```
tornare -1 che verra' interpretato come 255 e quindi un valore NON accettabile */
                        }
                        if (pid == 0)
                        {
                                /* codice del nipote */
                                printf("DEBUG-Sono il processo nipote del figlio di
indice %d e pid %d\n", i, getpid());
                                /* chiusura della pipe rimasta aperta di
comunicazione fra figlio-padre che il nipote non usa */
                                close(pipedFP[i][1]);
                                /* Chiusura delle pipe non usate nella
comunicazione con il padre  */
                                for (j=0; j < N; j++)
                                {
                                        close(pipedNP[j][0]);
                                        if (i != j) close(pipedNP[j][1]);
                                }

                                /* sia il figlio che il nipote devono aprire (con
due open separate per avere l'I/O pointer separato) lo stesso file in
lettura/scrittura, dato che dovranno operare delle trasformazioni */
                                if ((fd=open(argv[i+1], O_RDWR)) < 0)
                                {
                                        printf("Errore nella open del file %s da
parte del nipote\n", argv[i+1]);
                                        exit(-1); /* decidiamo, in caso di errore,
di tornare -1 che verra' interpretato come 255 e quindi un valore NON accettabile
*/
                                }
                                while (read(fd, &ch, 1))
                                {
                                        /* controlliamo se abbiamo trovato un
carattere alfabetico minuscolo */
                                        if (islower(ch))
                                        {
                                                /* questo carattere deve essere
trasformato nel corrispondente carattere alfabetico MAIUSCOLO e quindi per prima
cosa bisogna tornare indietro di una posizione */
                                                lseek(fd, -1L, SEEK_CUR);
                                                ch = ch - 32; /* trasformiamo il
carattere da minuscolo in MAIUSCOLO togliendo 32: volendo si poteva usare anche la
funzione di libreria toupper! */
                                                write(fd, &ch, 1); /* scriviamolo
sul file */
                                                /* e aggiorniamo il numero di
trasformazioni */
                                                trasformazioni++;
                                        }
                                        else
                                                ; /* non si deve fare nulla */
                                }
                                /* il nipote deve inviare al padre il numero di
trasformazioni operate */
                                write(pipedNP[i][1], &trasformazioni,
sizeof(trasformazioni));
                                /* torniamo il valore richiesto dal testo operando
una divisione intera per 256 */
                                ritorno=trasformazioni/256;
                                exit(ritorno);
```

```
                        }
                        /* codice figlio */
                        /* le pipe usate dal nipote vanno chiuse TUTTE */
                        /* ATTENZIONE: SOLO DOPO AVERE CREATO IL NIPOTE! */
                        for (j=0; j < N; j++)
                        {
                                close(pipedNP[j][0]);
                                close(pipedNP[j][1]);
                        }

                        /* il figlio ha un codice molto simile al nipote */
                        /* sia il figlio che il nipote devono aprire (con due open
separate per avere l'I/O pointer separato) lo stesso file in lettura/scrittura,
dato che dovranno operare delle trasformazioni */
                        if ((fd=open(argv[i+1], O_RDWR)) < 0)
                        {
                                printf("Errore nella open del file %s da parte del
figlio\n", argv[i+1]);
                                exit(-1); /* decidiamo, in caso di errore, di
tornare -1 che verra' interpretato come 255 e quindi un valore NON accettabile */
                        }
                        while (read(fd, &ch, 1))
                        {
                                /* controlliamo se abbiamo trovato un carattere
numerico */
                                if (isdigit(ch))
                                {
                                        /* questo carattere deve essere trasformato
nel carattere spazio e quindi per prima cosa bisogna tornare indietro di una
posizione */
                                        lseek(fd, -1L, SEEK_CUR);
                                        ch = ' '; /* trasformiamo il carattere
nello spazio */
                                        write(fd, &ch, 1); /* scriviamolo sul file
*/
                                        /* e aggiorniamo il numero di
trasformazioni */
                                        trasformazioni++;
                                }
                                else
                                        ; /* non si deve fare nulla */
                        }
                        /* il figlio deve inviare al padre il numero di
trasformazioni operate */
                        write(pipedFP[i][1], &trasformazioni,
sizeof(trasformazioni));
                        /* il figlio deve aspettare il nipote e stampare il suo pid
con il valore ritornato (come richiesto dal testo) */
                        if ((pid = wait(&status)) < 0)
                        {
                                printf("Errore in wait\n");
                                /* dato che il figlio deve tornare al padre un
proprio valore segnaliamo solo il problema */
                        }
                        if ((status & 0xFF) != 0)
                        {
                                printf("Nipote con pid %d terminato in modo
anomalo\n", pid);
                                /* dato che il figlio deve tornare al padre un
```

```
proprio valore segnaliamo solo il problema */
                        }
                        else
                                printf("Il nipote con pid=%d ha ritornato %d\n",
pid, ritorno=(int)((status >> 8) & 0xFF));

                        /* torniamo il valore richiesto dal testo operando una
divisione intera per 256 */
                        ritorno=trasformazioni/256;
                        exit(ritorno);
                }
        }

        /* Codice del padre */
        /* Il padre chiude i lati delle pipe che non usa */
        for (i=0; i < N; i++)
        {
                close(pipedFP[i][1]);
                close(pipedNP[i][1]);
        }

        /* Il padre recupera le informazioni dai figli e dai nipoti in ordine di
indice */
        for (i=0; i < N; i++)
        {
                read(pipedFP[i][0], &trasformazioni, sizeof(trasformazioni));
                printf("Il figlio di indice %d ha operato %ld trasformazioni di
caratteri numerici in carattere spazio sul file %s\n", i, trasformazioni,
argv[i+1]);
                read(pipedNP[i][0], &trasformazioni, sizeof(trasformazioni));
                printf("Il nipote di indice %d ha operato %ld trasformazioni di
caratteri minuscoli in MAIUSCOLI sullo stesso file %s\n", i, trasformazioni,
argv[i+1]);
        }

        /* Il padre aspetta i figli */
        for (i=0; i < N; i++)
        {
                if ((pid = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(7);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {
                        ritorno=(int)((status >> 8) &   0xFF);
                        if (ritorno==255)
                                printf("Il figlio con pid=%d ha ritornato %d e
quindi vuole dire che ci sono stati dei problemi\n", pid, ritorno);
                        else    printf("Il figlio con pid=%d ha ritornato %d\n",
pid, ritorno);
                }
        }
```

```
        exit(0);
}
```

# 8 SETTEMBRE 2021

La parte in C accetta un numero variabile di parametri N maggiore o uguale a 1 che rappresentano N nomi di file (F1, …FN). Il processo padre deve, per prima cosa, creare nella directory corrente un file fcreato con nome corrispondente al proprio cognome (tutto scritto in minuscolo, in caso di più cognomi se ne usi solo uno, inserendo un opportuno commento) e terminazione log (esempio, leonardi.log). Il processo padre deve generare un numero di processi figli pari a 2 ***N: i processi figli devono essere considerati a coppie e ogni coppia è costituita dai processi Pi***2 (detti processi pari) e Pi2+***1 (detti processi dispari) (con i che varia da 0 a N-1). Ogni coppia così determinata è associata ad uno dei file F1, …FN (in ordine): quindi ogni processo pari Pn (cioè con indice n pari) identificherà il file associato come Fn/2+1, mentre ogni processo dispari Pn (cioè con indice n dispari) identificherà il file associato come F(n+1)/2. Ogni processo pari è associato alle linee pari del proprio file e ogni processo dispari è associato alle linee dispari del proprio file. ATTENZIONE: Si consideri che la prima linea dei file abbia numero 1 e quindi sia dispari! Ognuno di tali processi figli Pn esegue concorrentemente e legge tutte le linee*** del proprio file associato: per ogni linea dispari, ogni processo figlio dispari Pn calcola la lunghezza della linea corrente compreso il terminatore di linea e la invia al padre e per ogni linea pari, ogni processo figlio pari Pn calcola la lunghezza della linea corrente compreso il terminatore di linea e la invia al padre. Il padre deve ricevere per ogni file F1, …FN da ognuna delle coppie tutte le lunghezze delle linee dispari/pari e le deve stampare su standard output in ordine di linea, come mostrato nell'esempio (riportato sul retro del foglio)! Al termine dell'esecuzione, ogni figlio Pn ritorna al padre il massimo delle lunghezze calcolate (sicuramente minore di 255); il padre deve scrivere sul file fcreato il PID di ogni figlio e il valore ritornato.

NOTA BENE NEL FILE C main.c SI USI OBBLIGATORIAMENTE:

- una variabile di nome N per il numero di file;
- una variabile di nome n per l'indice dei processi figli;
- una variabile di nome linea per la linea corrente (pari/dispari) letta dai figli dal proprio file;
- una variabile di nome nro per il valore massimo della lunghezza delle linee pari/dispari dei file;
- una variabile di nome fcreato per il file descriptor del file creato dal padre.

*Ogni linea si può supporre lunga 250 caratteri compreso il terminatore di linea.

**tags**: i figli sono 2N e comunicano con il padre, figli pari e dispari, figli a coppie

```c
/* Soluzione della Prova d'esame del 8 Settembre 2021: versione con un unico array
di 2N pipe */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

typedef int pipe_t[2];

int main(int argc, char **argv)
{
        /* -------- Variabili locali ---------- */
        int pid;                        /* process identifier per le altre fork()
*/
```

```c
        int N;                          /* numero di file passati sulla riga di
comando */
        int nro;                        /* massimo delle lunghezze delle linee
calcolate da ogni figlio */
        int nroLinea;                   /* numero delle linee calcolate da ogni
figlio e dal padre */
        int fcreato;                    /* file descriptor per creazione file da
parte del padre */
        int fd;                         /* file descriptor per apertura file */
        int status, ritorno;            /* variabili per la wait */
        pipe_t *piped;                  /* array dinamico di pipe descriptors per
comunicazioni figli-padre */
        int n;                          /* indice per i figli */
        int i, j;                       /* indici per i cicli */
        char linea[250];                /* array di caratteri per memorizzare la
linea: come indicato dal testo si puo' supporre una lunghezza massima di ogni linea
di 250 caratteri compreso il terminatore di linea */
        int nr1, nr2;                   /* variabili che vengono usate dal padre
per sapere se i figli hanno inviato qualcosa */
        int L1, L2;                     /* variabili che vengono usate dal padre
per recuperare le lunghezze inviate dai figli */
        /* ---------------------------------- */

        /* Controllo sul numero di parametri */
        if (argc < 2) /* Meno di un parametro */
        {
                printf("Errore nel numero dei parametri\n");
                exit(1);
        }

        /* Calcoliamo il numero di file passati */
        N = argc - 1;
        printf("Sono il padre con pid %d e %d file e creero' %d processi figli\n",
getpid(), N, 2*N);

        /* PER PRIMA COSA creazione file con mio cognome e terminazione log */
        if ((fcreato=creat("leonardi.log", 0644)) < 0)
        {
                printf("Errore nella creat del file %s\n", "leonardi.log");
                exit(2);
        }

        /* Allocazione dell'array di 2*N pipe descriptors*/
        piped = (pipe_t *) malloc (2*N*sizeof(pipe_t));
        if (piped == NULL)
        {
                printf("Errore nella allocazione della memoria\n");
                exit(3);
        }

        /* Creazione delle N pipe figli-padre */
        for (n=0; n < 2*N; n++)
        {
                if(pipe(piped[n]) < 0)
                {
                        printf("Errore nella creazione della pipe\n");
                        exit(4);
                }
        }
```

```
        /* Ciclo di generazione dei figli: NOTA BENE DEVONO ESSERE 2 * N  */
        for (n=0; n < 2*N; n++)
        {
                if ( (pid = fork()) < 0)
                {
                        printf("Errore nella fork del figlio %d-esimo\n", n);
                        exit(5);
                }

                if (pid == 0)
                {
                        /* codice del figlio */
                        /* dopo la consegna, ci si e' accorti che i vari processi
potevano direttamente individuare il file associato usando l'indice n/2+1 senza
bisogno di distinguere caso indice dispari o pari e quindi la soluzione riporta
questa semplificazione; chiaramente le consegne che prevedevano codice
differenziato sono state considerate corrette! */
                        printf("Sono il processo figlio di indice %d e pid %d e
sono associato al file %s\n", n, getpid(), argv[n/2+1]);

                        /* Chiusura delle pipe non usate nella comunicazione con il
padre */
                        for (j=0; j < 2*N; j++)
                        {
                                close(piped[j][0]);
                                if (n != j) close(piped[j][1]);
                        }

                        if ((fd=open(argv[n/2+1], O_RDONLY)) < 0)
                        {
                                printf("Errore nella open del file %s\n",
argv[n/2+1]);
                                exit(-1); /* in caso di errore nei figli decidiamo
di tornare -1 che corrispondera' per il padre al valore 255 che supponiamo non
essere un valore accettabile di ritorno */
                        }

                        /* adesso il figlio legge dal file una linea alla volta */
                        j=0;            /* azzeriamo l'indice della linea */
                        nroLinea=0;     /* azzeriamo il numero della linea */
                        nro=-1;         /* settiamo il massimo a -1 */
                        while (read(fd, &(linea[j]), 1))
                        {
                                if (linea[j] == '\n')
                                {
                                        /* dobbiamo mandare al padre la lunghezza
della linea selezionata compreso il terminatore di linea (come int) e quindi
incrementiamo j */
                                        j++;
                                        nroLinea++; /* la prima linea sarÃ  la
numero 1! */
                                        if ( ((n%2) != 0) && ((nroLinea%2) != 0) )
/* processo associato a linea dispari e numero di linea dispari */
                                        {
                                                write(piped[n][1], &j, sizeof(j));
                                                //printf("DEBUG PROCESSO FIGLIO DI
INDICE DISPARI n=%d ha scritto sulla pipe di indice %d il valore %d\n", n, n, j);
                                                /* verifichiamo e nel caso
```

```
aggiorniamo il massimo */
                                        if (j > nro)
                                                nro=j;
                                }
                                if ( ((n%2) == 0) && ((nroLinea%2) == 0) )
/* processo associato a linea pari e numero di linea pari */
                                {
                                        write(piped[n][1], &j, sizeof(j));
                                        //printf("DEBUG PROCESSO FIGLIO DI
INDICE PARI n=%d ha scritto sulla pipe di indice %d il valore %d\n", n, n, j);
                                        /* verifichiamo e nel caso
aggiorniamo il massimo */
                                        if (j > nro)
                                                nro=j;
                                }

                                j=0; /* azzeriamo l'indice per le prossime
linee */
                        }
                        else j++; /* continuiamo a leggere */
                }
                /* ogni figlio deve ritornare al padre il valore
corrispondente al massimo */
                exit(nro);
                }
        }

        /* Codice del padre */
        /* Il padre chiude i lati delle pipe che non usa */
        for (n=0; n < 2*N; n++)
                close(piped[n][1]);

        /* Il padre recupera le informazioni dai figli: prima in ordine di file e
quindi di linee */
        for (i=0; i < N; i++)
        {
                nroLinea=1; /* la prima linea ha numero 1 */
                printf("Le lunghezze delle linee del file %s sono:\n", argv[i+1]);
                do
                {
                        /* il padre recupera le lunghezze delle linee da ogni
figlio dispari e pari */
                        nr1=read(piped[i*2+1][0], &L1, sizeof(L1));
                        nr2=read(piped[i*2][0], &L2, sizeof(L2));
                        //printf("DEBUG PADRE linea numero %d e nr1=%d da pipe di
indice %d e nr2=%d da pipe di indice %d\n", nroLinea, nr1,i*2+1, nr2, i*2);
                        if (nr1 != 0)
                        {
                                printf("linea numero %d e' lunga %d\n", nroLinea,
L1);
                                nroLinea++;             /* incrementiamo il numero
di linea */
                        }
                        if (nr2 != 0)
                        {
                                printf("linea numero %d e' lunga %d\n", nroLinea,
L2);
                                nroLinea++;             /* incrementiamo il numero
di linea */
```

```
                        }

                } while (nr1||nr2);
                /* in alternativa si poteva fare un while(1) ed inserire dei break
quando la nr1 o nr2 arrivano a 0 */
        }

        /* Il padre aspetta tutti i figli */
        for (n=0; n < 2*N; n++)
        {
                pid = wait(&status);
                if (pid < 0)
                {
                printf("Errore in wait\n");
                exit(9);
                }

                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pid);
                else
                {
                        ritorno=(int)((status >> 8) &   0xFF);
                        /* dobbiamo preparare la stringa da scrivere sul file
creato: N.B. usiamo sprintf e linea che tanto il padre non ha usato */
                        sprintf(linea,"Il figlio con pid=%d ha ritornato %d (se 255
significa che il figlio e' terminato con un errore)\n", pid, ritorno);
                        write(fcreato, linea, strlen(linea));
                }
        }
        exit(0);
}
```

# 11 SETTEMBRE 2019

La parte in C accetta un numero variabile pari N di parametri (con N pari e maggiore o uguale a 2, da controllare) che rappresentano N nomi di file (F1, F2. ... FN): la lunghezza in caratteri dei file è la stessa (minore di 255, questo viene garantito dalla parte shell e NON deve essere controllato). Il processo padre deve generare N processi figli Pi (P0 … PN-1): i processi figli Pi (con i che varia da 0 a N-1) sono associati agli N file Ff (con f= i+1). I figli si devono considerare a coppie ordinate: ogni coppia è costituita dal processo di indice pari Pp (i = 0, 2, ... NM»-2) e dal processo di indice dispari Pd (i = 1, 3, ... NM»-1). La comunicazione in ognuna di tali coppie deve essere dal processo Pp al processo Pd; la prima azione che dovrà compiere il processo Pd sarà quella di creare un file (Fcreato) il cui nome sia la concatenazione del nome del file associato Ff con la stringa " .MAGGIORE". Ogni processo figlio esegue concorrentemente leggendo i caratteri del file ad esso associato Ff: dopo la lettura di ogni carattere (Cp) da parte del processo Pp, questo viene comunicato al processo Pd, il quale lo riceve dopo aver letto il proprio carattere (Cd); il processo Pd deve confrontare il proprio carattere Cd con il carattere ricevuto Cp e se Cd risulta strettamente maggiore di Cp, lo scrive sul file Fcreato, altrimenti scrive Cp. Ogni processo figlio deve ritornare al padre il numero di caratteri letti dal proprio file. Il padre, dopo che i figli sono terminati, deve stampare su standard output i PID di ogni figlio con il corrispondente valore ritornato. Il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

**tags**: i figli comunicano tra di loro, il padre riceve da ogni figlio una singola informazione, figli pari e dispari, figli a coppie

```c
/* soluzione parte C esame dell'11 Settembre 2019: questa soluzione considera che
la comunicazione in ogni coppia sia dal processo pari al processo dispari e che il
processo dispari deve creare il file con terminazione ".MAGGIORE" */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define PERM 0644

typedef int pipe_t[2];          /* tipo di dato per contenere i file descriptors di
una pipe */

int main(int argc, char **argv)
{
        int N;                  /* numero di file: i processi figli saranno il
doppio! */
        int pid;                /* variabile per fork */
        pipe_t *pipe_f;         /* array di pipe per la comunicazione dai figli
pari e figli dispari */
        int fd;                 /* variabile per open */
        char *Fcreato;          /* variabile per nome file da creare da parte dei
processi dispari */
        int fdw;                /* variabile per creat */
        char ch;                /* variabile per leggere dai figli */
        char ch1;               /* variabile per ricevere dal processo-coppia */
        int letti;              /* variabile per tenere traccia del numero di
caratteri letti */
        int status, pidFiglio, ritorno; /* per wait */
        int i, j;               /* indici per cicli */
        int nr, nw;             /* per controlli read e write su/da pipe */

/* Controllo sul numero di parametri */
if ( (argc < 3) && (argc - 1) % 2 !=0)
{
        printf("Errore numero parametri %d\n", argc);
        exit(1);
}

/* calcoliamo il numero dei file */
N = argc - 1;

printf("Numero processi da creare %d\n", N);

/* allocazione memoria dinamica per pipe_fi. NOTA BENE: servono un numero di pipe
che e' la meta' del numero di figli! */
pipe_f=malloc(N/2*sizeof(pipe_t));
if (pipe_f == NULL)
{
        printf("Errore nelle malloc\n");
        exit(2);
}

/* creazione delle pipe: ATTENZIONE VANNO CREATE N/2 pipe */
```

```
for (i=0; i < N/2; i++)
{
        if (pipe(pipe_f[i])!=0)
        {
                printf("Errore creazione delle pipe\n"); exit(3);
        }
}

/* creazione dei processi figli: ne devono essere creati N */
for (i=0; i < N; i++)
{
        pid=fork();
        if (pid < 0)  /* errore */
        {
                printf("Errore nella fork con indice %d\n", i);
                exit(4);
        }
        if (pid == 0)
        {
                /* codice del figlio */
                /* stampa di debugging */
                printf("Figlio di indice %d e pid %d associato al file
%s\n",i,getpid(),argv[i+1]);
                /* ogni figlio deve aprire il suo file associato */
                fd=open(argv[i+1], O_RDONLY);
                if (fd < 0)
                {
                        printf("Impossibile aprire il file %s\n", argv[i+1]);
                        exit(0); /* in caso di errore torniamo 0 che non e' un
valore accettabile (per quanto risulta dalla specifica della parte shell) */
                }

                /* inizializziamo letti */
                letti=0;
                if (i % 2 == 0) /* siamo nel codice dei figli pari */
                {
                        /* chiudiamo le pipe che non servono */
                        /* ogni figlio pari scrive solo su pipe_f[i] */
                        for (j=0;j<N/2;j++)
                        {
                                close(pipe_f[j][0]);
                                if (j!=i/2)     /* ATTENZIONE ALL'INDICE CHE DEVE
ESSERE USATO */
                                {
                                        close(pipe_f[j][1]);
                                }
                        }

                        while (read(fd, &ch, 1))
                        {
                                /* incrementiamo letti */
                                letti++;
                                /* ad ogni carattere letto da un processo di indice
pari, bisogna mandare il carattere al processo-coppia: l'indice della pipe da usare
si trova dividendo i per 2  */
                                nw=write(pipe_f[i/2][1], &ch, sizeof(ch));
                                if (nw != sizeof(ch))
                                {
                                        printf("Impossibile scrivere sulla pipe per
```

```
il processo di indice %d\n", i);
                                        exit(0);
                                }
                        }
                }
                else /* siamo nel codice dei figli dispari */
                {
                        /* chiudiamo le pipe che non servono */
                        /* ogni figlio dispari legge solo da pipe_f[i] */
                        for (j=0;j<N/2;j++)
                        {
                                close(pipe_f[j][1]);
                                if (j!= i/2)     /* ATTENZIONE ALL'INDICE CHE DEVE
ESSERE USATO */
                                {
                                        close(pipe_f[j][0]);
                                }
                        }

                        /* i figli dispari devono creare il file specificato */
                        Fcreato=(char *)malloc(strlen(argv[i+1]) + 10); /* bisogna
allocare una stringa lunga come il nome del file + il carattere '.' + i caratteri
della parola MAGGIORE (8) + il terminatore di stringa */
                        if (Fcreato == NULL)
                        {
                                printf("Errore nelle malloc\n");
                                exit(0);
                        }
                        /* copiamo il nome del file associato al figlio dispari */
                        strcpy(Fcreato, argv[i+1]);
                        /* concateniamo la stringa specificata dal testo */
                        strcat(Fcreato,".MAGGIORE");
                        fdw=creat(Fcreato, PERM);
                        if (fdw < 0)
                        {
                                printf("Impossibile creare il file %s\n", Fcreato);
                                exit(0);
                        }

                        while (read(fd, &ch, 1))
                        {
                                /* incrementiamo letti */
                                letti++;
                                /* ad ogni carattere letto da un processo di indice
dispari, bisogna ricevere il carattere letto dal processo-coppia: l'indice della
pipe da usare si trova dividendo i per 2  */
                                nr=read(pipe_f[i/2][0], &ch1, sizeof(ch1));
                                if (nr != sizeof(ch1))
                                {
                                        printf("Impossibile leggere dalla pipe per
il processo di indice %d\n", i);
                                        exit(0);
                                }
                                /* printf("Caratteri letti da processo di indice
%d: ch = %c e ch1 = %c\n", i, ch, ch1); */
                                if (ch <= ch1)          /* se il carattere letto e'
minore o uguale al carattere ricevuto, deve essere scritto il carattere ricevuto,
altrimenti il carattere letto */
                                {
```

```
                                        ch = ch1;
                                        /* printf("Carattere che verra' scritto da
processo di indice %d: ch = %c\n", i, ch); */
                                }
                                write(fdw, &ch, 1);

                        }
                }
                exit(letti); /* torniamo il numero di caratteri letti (supposto dal
testo della parte shell <= di 255) */
        }
}

/*codice del padre*/
/* chiudiamo tutte le pipe, dato che le usano solo i figli */
for (i=0;i<N;i++)
{
   close(pipe_f[i][0]);
   close(pipe_f[i][1]);
}

/* Attesa della terminazione dei figli */
for(i=0;i<N;i++)
{
   pidFiglio = wait(&status);
   if (pidFiglio < 0)
   {
      printf("Errore wait\n");
      exit(5);
   }
   if ((status & 0xFF) != 0)
                printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
   else
   {
                ritorno=(int)((status >> 8) & 0xFF);
                printf("Il figlio con pid=%d ha ritornato %d (se 0 problemi!)\n",
pidFiglio, ritorno);
   }
}
exit(0);
}/* fine del main */
```

# 19 GENNAIO 2022

La parte in C accetta un numero variabile di parametri N+1 con N maggiore o uguale a 1: i primi N rappresentano nomi di file (F1, …FN), mentre l'ultimo parametro C rappresenta un numero intero strettamente positivo e dispari (da controllare): si può ipotizzare che la lunghezza di tutti i file sia uguale, pari e multiplo intero di C (senza verificarlo). Il processo padre deve generare 2*N processi figli (P0 … P2*N- 1); tali processi figli costituiscono N coppie di processi: ogni coppia Ci è composta dal processo Pi (primo processo della coppia) e dal processo Pi+N (secondo processo della coppia), con i variabile da 0 a N-1. Ogni coppia di processi figli Ci è associata ad uno dei file Fi+1*. Il secondo processo della coppia deve creare un file il cui nome risulti dalla concatenazione del nome del file associato alla coppia con la stringa .mescolato (ad esempio se il file associato è /tmp/pippo.txt, il file creato si deve chiamare /tmp/pippo.txt.mescolato). Tutte le coppie Ci di processi figli eseguono concorrentemente leggendo il proprio file associato: in particolare, il primo processo di ogni coppia deve leggere la prima metà del file associato, mentre il secondo processo la seconda metà del file; inoltre, per entrambi i processi di ogni

coppia la lettura deve avvenire a blocchi di dati di grandezza uguale a C byte. Il secondo processo di ogni coppia, dopo la lettura di ogni blocco di dati B2 (con un'unica read!) della sua seconda metà del file, lo scrive (con un'unica write!) sul file creato; quindi deve ricevere (con un'unica read!) dal primo processo della coppia il suo corrispondente blocco di dati B1 e quindi deve scriverlo (sempre con un'unica write!) sul file creato; viceversa, il primo processo di ogni coppia, dopo la lettura di ogni blocco di dati B1 (con un'unica read!) della sua prima metà del file, lo comunica (con un'unica write!) al secondo processo della coppia (si veda un esempio riportato sul retro del foglio). Al termine, ogni processo di ogni coppia deve ritornare al padre il numero di blocchi (nro) letti dalla propria metà del file. Il padre, dopo che i figli sono terminati, deve stampare su standard output i PID di ogni figlio con il corrispondente valore ritornato.

NOTA BENE NEL FILE C main.c SI USI OBBLIGATORIAMENTE:

- una variabile di nome N per il numero di file;
- una variabile di nome i per l'indice dei processi figli;
- una variabile di nome b per il blocco corrente (B1 o B2) letto dai figli dal file;
- una variabile di nome nro per il numero di blocchi letti dalla propria metà del file;
- una variabile di nome fcreato per il file descriptor del file creato dal secondo processo di ogni coppia.

*Se N è 3 (i varia da 0 a 2), le coppie di processi e i file associati sono P0-P3 per F1, P1-P4 per F2 e P2-P5 per F3.

**tags**: i figli sono 2N e comunicano tra di loro, Figlio Pi con Pi+N, figli a coppie

```c
/* soluzione parte C esame del 19 Gennaio 2022: la comunicazione in ogni coppia va
dal primo processo della coppia al secondo processo della coppia ed e' il secondo
processo della coppia deve creare il file con terminazione ".mescolato" sul quale
poi deve scrivere */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define PERM 0644

typedef int pipe_t[2];          /* tipo di dato per contenere i file descriptors di
una pipe */

int main(int argc, char **argv)
{
        int N;                  /* numero di file: i processi figli saranno il
doppio! */
        /* N nome specificato nel testo */
        int C;                  /* numero intero positivo dispari */
        int pid;                /* variabile per fork */
        pipe_t *pipe_ps;        /* array di pipe per la comunicazione dai figli
primi della coppia ai figli secondi della coppia */
        int fd;                 /* variabile per open */
        char *FCreato;          /* variabile per nome file da creare da parte dei
processi figli secondi della coppia */
        int fcreato;            /* variabile per creat */
        /* fcreato nome specificato nel testo */
        char *b;                /* variabile per leggere dai figli */
        /* b nome specificato nel testo */
        int nroTotale;          /* variabile per tenere traccia del numero di
```

```
blocchi presenti nei file */
        int nro;                /* variabile per tenere traccia del numero di
blocchi leti dalla propria meta' del file */
        /* nro nome specificato nel testo */
        int i, j;               /* indici per cicli */
        /* i nome specificato nel testo */
        int nr, nw;             /* per controllo su read/write */
        int status, pidFiglio, ritorno; /* per wait */

        /* Controllo sul numero di parametri: N deve essere maggiore o uguale a 1
*/
        if (argc < 3)
        {
                printf("Errore nel numero dei parametri dato che argc=%d (bisogna
passare almeno un nome di file e un numero!)\n", argc);
                exit(1);
        }

        /* controllo sull'ultimo parametro: numero dispari C */
        C = atoi(argv[argc-1]);
        if ( (C <= 0) || (C % 2 == 0) )
        {
                printf("Errore numero C: non strettamente positivo o non dispari
%s\n", argv[argc-1]);
                exit(2);
        }

        /* calcoliamo il numero dei file */
        N = argc - 2;

        printf("DEBUG-Numero processi da creare %d con C=%d\n", 2*N, C);

        /* allocazione memoria dinamica per buffer */
        b=(char *)malloc(C*sizeof(char));
        if (b == NULL)
        {
                printf("Errore nella malloc per buffer b\n");
                exit(3);
        }

        /* allocazione memoria dinamica per pipe_ps. NOTA BENE: servono un numero
di pipe che e' la meta' del numero di figli e quindi solo N! */
        pipe_ps=(pipe_t *)malloc(N*sizeof(pipe_t));
        if (pipe_ps == NULL)
        {
                printf("Errore nella malloc per le pipe\n");
                exit(4);
        }

        /* creazione delle pipe: ATTENZIONE VANNO CREATE solo N pipe */
        for (i=0; i < N; i++)
        {
                if (pipe(pipe_ps[i])!=0)
                {
                        printf("Errore creazione delle pipe\n");
                        exit(5);
                }
        }
```

```
        /* Ciclo di generazione dei figli: ne devono essere creati 2*N */
        for (i=0; i < 2*N; i++)
        {
                pid=fork();
                if (pid < 0)  /* errore */
                {
                        printf("Errore nella fork con indice %d\n", i);
                        exit(6);
                }
                if (pid == 0)
                {
                        /* codice del figlio: in caso di errore torniamo 0 che non
e' un valore accettabile (per quanto risulta dalla specifica della parte shell) */
                        if (i < N) /* siamo nel codice dei figli primi della coppia
*/
                        {
                                /* stampa di debugging */
                                printf("DEBUG-PRIMO DELLA COPPIA-Figlio di indice
%d e pid %d associato al file %s\n",i,getpid(),argv[i+1]);
                                /* chiudiamo le pipe che non servono */
                                /* ogni figlio PRIMO della coppia scrive solo sulla
pipe_ps[i] */
                                for (j=0;j<N;j++)
                                {
                                        close(pipe_ps[j][0]);
                                        if (j!=i)
                                        {
                                                close(pipe_ps[j][1]);
                                        }
                                }

                                /* ogni figlio deve aprire il suo file associato */
                                fd=open(argv[i+1], O_RDONLY);
                                if (fd < 0)
                                {
                                        printf("Impossibile aprire il file %s\n",
argv[i+1]);
                                        exit(0); /* in caso di errore, decidiamo di
tornare 0 che non e' un valore accettabile */
                                }

                                /* calcoliamo la lunghezza in blocchi del file */
                                nroTotale = lseek(fd, 0L, 2) / C;
                                nro=nroTotale/2;        /* ogni figlio legge meta'
del file */
                                /* bisogna riportare l'I/0 pointer all'inizio del
file */
                                lseek(fd, 0L, 0);
                                for (j=0;j<nro;j++)
                                {
                                        read(fd, b, C);
                                        /* ogni blocco letto dal PRIMO processo
della coppia deve essere inviato al processo SECONDO della coppia */
                                        nw=write(pipe_ps[i][1], b, C);
                                        if (nw != C)
                                        {
                                                printf("Errore in scrittura su pipe
%d\n", i);
                                                exit(0);
```

```
                                }
                        }
                }
                else /* siamo nel codice dei figli secondi della coppia */
                {
                        /* stampa di debugging */
                        printf("DEBUG-SECONDO DELLA COPPIA-Figlio di indice
%d e pid %d associato al file %s\n",i,getpid(),argv[i-N+1]);
                        /* i figli secondi della coppia devono creare il
file specificato */
                        FCreato=(char *)malloc(strlen(argv[i-N+1]) + 11);
/* bisogna allocare una stringa lunga come il nome del file associato + il
carattere '.' + i caratteri della parola mescolato (9) + il terminatore di stringa:
ATTENZIONE ALL'INDICE PER INDIVIDUARE IL FILE */
                        if (FCreato == NULL)
                        {
                                printf("Errore nella malloc\n");
                                exit(0);
                        }
                        /* copiamo il nome del file associato */
                        strcpy(FCreato, argv[i-N+1]);
                        /* concateniamo la stringa specificata dal testo */
                        strcat(FCreato,".mescolato");
                        fcreato=creat(FCreato, PERM);
                        if (fcreato < 0)
                        {
                                printf("Impossibile creare il file %s\n",
FCreato);
                                exit(0);
                        }

                        /* chiudiamo le pipe che non servono */
                        /* ogni figlio SECONDO della coppia legge solo da
pipe_ps[i-N] */
                        for (j=0;j<N;j++)
                        {
                                close(pipe_ps[j][1]);
                                if (j!= i-N)     /* ATTENZIONE ALL'INDICE
CHE DEVE ESSERE USATO */
                                {
                                        close(pipe_ps[j][0]);
                                }
                        }

                        /* ogni figlio deve aprire il suo file associato:
siamo nei figli secondi della coppia e quindi attenzione all'indice */
                        fd=open(argv[i-N+1], O_RDONLY);
                        if (fd < 0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[i-N+1]);
                                exit(0);
                        }

                        /* calcoliamo la lunghezza in blocchi del file */
                        nroTotale = lseek(fd, 0L, 2) / C;
                        nro=nroTotale/2;        /* ogni figlio legge meta'
del file */
                        /* bisogna posizionare l'I/0 pointer a meta' del
```

```
file */
                                lseek(fd, (long)nro * C, 0);
                                for (j=0;j<nro;j++)
                                {
                                        read(fd, b, C);
                                        /* ogni blocco letto dal processo SECONDO
della coppia, bisogna scriverlo sul file */
                                        write(fcreato, b, C);
                                        /* dobbiamo a questo punto aspettare il
blocco dal processo PRIMO della coppia: attenzione all'indice */
                                        nr=read(pipe_ps[i-N][0], b, C);
                                        if (nr != C)
                                        {
                                                printf("Errore in lettura da pipe
%d\n", i-N);
                                                exit(0);
                                        }
                                        /* ogni blocco ricevuto dal processo PRIMO
della coppia, bisogna scriverlo sul file */
                                        write(fcreato, b, C);
                                }
                        }
                        exit(nro); /* torniamo il numero di blocchi letti (supposto
<= di 255) */
                }
        }

        /*codice del padre*/
        /* chiudiamo tutte le pipe, dato che le usano solo i figli */
        for (i=0;i<N;i++)
        {
                close(pipe_ps[i][0]);
                close(pipe_ps[i][1]);
        }

        /* Attesa della terminazione dei figli */
        for(i=0;i<2*N;i++)
        {
                pidFiglio = wait(&status);
                if (pidFiglio < 0)
                {
                        printf("Errore wait\n");
                        exit(7);
                }
                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 0
problemi)\n", pidFiglio, ritorno);
                }
        }

        exit(0);
}/* fine del main */
```

# 10 LUGLIO 2019

La parte in C accetta un numero variabile N+1 di parametri (con N maggiore o uguale a 2, da controllare) che rappresentano i primi N nomi di file (F1, F2. ... FN), mentre l'ultimo rappresenta un singolo carattere Cz (da controllare). Il processo padre deve generare 2 * N processi figli (P0, P1 .. PN-1, PN, PN+1 … P2*N-1): i processi figli vanno considerati a coppie, ognuna delle quali è associata ad uno dei file Ff (con f = i+1, con i che varia da 0 a … N-1 e che corrisponde al file Fx con x =* 2N-i per i da N a 2*N-1). In particolare, la prima coppia è costituita dal processo P0 e dal processo P2*N-1, la seconda dal processo P1 e dal processo P2*N-2 e così via fino alla coppia costituita dal processo PN-1 e dal processo PN. Entrambi i processi della coppia (nel seguito chiamati primo e secondo processo) devono cercare il carattere Cz nel file associato Fi sempre fino alla fine attuando una sorta di staffetta così come illustrato nel seguito. Il primo processo della coppia Pi (con i che varia da 0 a … N-1) deve cominciare a leggere dal file associato Ff cercando la prima occorrenza del carattere Cz; appena trovata deve comunicare al secondo processo della coppia Pi (con i da N a* 2N-1) la posizione del carattere trovato all'interno del file (in termini di long int); quindi il secondo processo della coppia deve partire nello stesso file associato Fx (che corrisponde a Ff) con la sua ricerca del carattere Cz dalla posizione seguente a quella ricevuta; appena trovata una nuova occorrenza di Cz, deve comunicare al primo processo della coppia la posizione del carattere trovato all'interno del file (in termini di long int) che quindi riparte dalla posizione seguente a cercare; tale staffetta deve avere termine quando il file è finito* . Al termine, ogni processo figlio deve ritornare al padre il numero di occorrenze del carattere Cz trovate dal singolo processo della coppia (supposto minore o uguale a 255) e il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

*Si precisa che è garantito dalla parte Shell che nel file vengano trovate almeno DUE occorrenze del carattere Cz e quindi la staffetta viene eseguita almeno una volta dal primo processo al secondo e almeno una volta dal secondo processo al primo. SI FACCIA ATTENZIONE ALL'INDICE CHE DEVE ESSERE USATO DAL SECONDO PROCESSO PER INDIVIDUARE IL FILE SU CUI OPERARE COSÌ ALL'INDICE CHE DEVE ESSERE USATO PER LA PIPE DA USARE IN LETTURA E PER LA PIPE DA USARE IN SCRITTURA!

**tags**: Figlio Pi con p2N-1-i, i figli comunicano tra di loro, figli a coppie

```c
/* soluzione parte C esame del 10 Luglio 2019 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

typedef int pipe_t[2];          /* tipo di dato per contenere i file descriptors di
una pipe */

int main(int argc, char **argv)
{
        int N;                  /* numero di file: i processi figli saranno il
doppio! */
        int pid;                /* variabile per fork */
        pipe_t *pipe_f;         /* array di pipe per la comunicazione dai primi N
figli agli ultimi N figli */
        pipe_t *pipe_fbis;      /* array di pipe per la comunicazione dagli ultimi
N figli ai primi N figli */
        /* OSSERVAZIONE: IN ALTERNATIVA POTEVA ESSERE USATO UN SOLO ARRAY DI
DIMENSIONE 2 * N: CHIARAMENTE IN QUESTO CASO GLI INDICI DA USARE SAREBBERO DIVERSI
RISPETTO A QUESTA SOLUZIONE! */
```

```c
        int fd;                 /* variabile per open */
        char ch;                /* variabile per leggere dai figli */
        char Cz;                /* variabile per tenere traccia del carattere da
cercare */
        int occ;                /* variabile per tenere traccia del numero di
occorrenze trovate */
        long int pos;           /* posizione corrente del carattere trovato:
inviamo il valore ricavato dalla lseek decrementato di 1 dato che dopo la lettura
l'I/O pointer e' posizionato sul carattere seguente quello letto */
        long int posLetta;      /* posizione corrente del carattere trovato
ricevuta */
        int status, pidFiglio, ritorno; /* per wait */
        int i, j;               /* indici per cicli */
        int nr, nw;             /* per controlli read e write su/da pipe */

/* Controllo sul numero di parametri */
if (argc < 4)
{
        printf("Errore numero parametri %d\n", argc);
        exit(1);
}

/* calcoliamo il numero dei file: ATTENZIONE BISOGNA TOGLIERE 2 PERCHE' C'E' ANCHE
IL CARATTERE Cz */
N = argc - 2;

/* Controlliamo se l'ultimo parametro e' un singolo carattere */
if (strlen(argv[argc-1]) != 1)
{
        printf("Errore ultimo parametro non singolo carattere %s\n", argv[argc-1]);
        exit(2);
}

Cz = argv[argc-1][0]; /* isoliamo il carattere che devono cercare i figli */
printf("Carattere da cercare %c\n", Cz);

/* allocazione memoria dinamica per pipe_f e pipe_fbis */
pipe_f=malloc(N*sizeof(pipe_t));
pipe_fbis=malloc(N*sizeof(pipe_t));
if ((pipe_f == NULL) || (pipe_fbis == NULL))
{
        printf("Errore nelle malloc\n");
        exit(3);
}

/* creazione delle pipe */
for (i=0; i < N; i++)
{
        if (pipe(pipe_f[i])!=0)
        {
                printf("Errore creazione delle pipe primi N figli e gli ultimi
N\n");
                exit(4);
        }
        if (pipe(pipe_fbis[i])!=0)
        {
                printf("Errore creazione delle pipe ultimi N figli e i primii
N\n");
                exit(5);
```

```
        }
}

/* creazione dei processi figli: ne devono essere creati 2 * N */
for (i=0; i < 2*N; i++)
{
        pid=fork();
        if (pid < 0)  /* errore */
        {
                printf("Errore nella fork con indice %d\n", i);
                exit(6);
        }
        if (pid == 0)
        {
                /* codice del figlio */
                /* stampa di debugging */
                if (i < N) /* siamo nel codice dei primi N figli */
                {
                        printf("Figlio di indice %d e pid %d associato al file
%s\n",i,getpid(), argv[i+1]);
                        /* chiudiamo le pipe che non servono */
                        /* ogni figlio scrive solo su pipe_f[i] e legge solo da
pipe_fbis[i] */
                        for (j=0;j<N;j++)
                        {
                                close(pipe_f[j][0]);
                                close(pipe_fbis[j][1]);
                                if (j!=i)
                                {
                                        close(pipe_f[j][1]);
                                        close(pipe_fbis[j][0]);
                                }
                        }

                        /* per i primi N processi, il file viene individuato come
al solito */
                        fd=open(argv[i+1], O_RDONLY);
                        if (fd < 0)
                        {
                                printf("Impossibile aprire il file %s\n",
argv[i+1]);
                                exit(0); /* in caso di errore torniamo 0 che non e'
un valore accettabile (per quanto risulta dalla specifica della parte shell) */
                        }

                        /* inizializziamo occ */
                        occ=0;
                        while (read(fd, &ch, 1))
                        {
                                if (ch == Cz) /* se abbiamo trovato il carattere da
cercare */
                                {
                                        /* incrementiamo occ */
                                        occ++;
                                        /* calcoliamo la posizione del carattere */
                                        /* il valore ricavato dalla lseek lo
decrementiamo di 1 dato che dopo la lettura l'I/O pointer e' posizionato sul
carattere seguente quello letto */
                                        pos=lseek(fd, 0L, SEEK_CUR) - 1;
```

```
                                        //printf("DEBUG- VALORE DI pos %ld per
processo di indice %d che sto per mandare su pipe_f[i][1] %d\n", pos, i, pipe_f[i]
[1]);
                                        /* inviamo la posizione del carattere
all'altro processo della coppia */
                                        nw=write(pipe_f[i][1], &pos, sizeof(pos));
                                        if (nw != sizeof(pos))
                                        {
                                                printf("Impossibile scrivere sulla
pipe per il processo di indice %d\n", i);
                                                exit(0);
                                        }
                                        /* aspettiamo dall'altro processo della
coppia la nuova posizione da cui si deve riprendere la ricerca */
                                        nr=read(pipe_fbis[i][0], &posLetta,
sizeof(posLetta));
                                        //printf("DEBUG- VALORE DI nr %d per
processo di indice %d\n", nr, i);
                                        if (nr != sizeof(posLetta))
                                        {
                                                /* se non mi viene inviato alcuna
posizione vuole dire che l'altro processo della coppia NON ha trovato altre
occorrenze e quindi si puÃ² terminare la lettura */
                                                break;
                                        }
                                        /* printf("DEBUG- VALORE DI pos %ld per
processo di indice %d che ho ricevuto da pipe_fbis[i][0] %d\n", pos, i,
pipe_fbis[i][0]); */
                                        /* spostiamo l'I/O pointer nella posizione
seguente! */
                                        lseek(fd, posLetta+1, SEEK_SET);
                                }
                                else
                                {  /* nulla, si continua a leggere */
                                        ;
                                }
                        }
                        exit(occ); /* torniamo il numero di occorrenze trovate
(supposto dal testo <= di 255) */
                }
                else /* siamo nel codice degli ultimi N figli */
                {
                        printf("SECONDA SERIE DI FIGLI-Figlio di indice %d e pid %d
associato al file %s\n",i,getpid(), argv[2*N-i]); /* ATTENZIONE ALL'INDICE CHE DEVE
ESSERE USATO */
                        /* chiudiamo le pipe che non servono */
                        /* ogni figlio scrive solo su pipe_fbis[i] e legge solo da
pipe_f[i] */
                        for (j=0;j<N;j++)
                        {
                                close(pipe_f[j][1]);
                                close(pipe_fbis[j][0]);
                                if (j!= 2*N-i-1)        /* ATTENZIONE ALL'INDICE
CHE DEVE ESSERE USATO */
                                {
                                        close(pipe_f[j][0]);
                                        close(pipe_fbis[j][1]);
                                }
                        }
```

```
                        /* per gli ultimi N processi, il file viene individuato
come indicato nel testo! */
                        fd=open(argv[2*N-i], O_RDONLY);
                        if (fd < 0)
                        {
                                printf("Impossibile aprire il file %s\n", argv[2*N-
i]);
                                exit(0); /* in caso di errore torniamo 0 che non e'
un valore accettabile (per quanto risulta dalla specifica della parte shell) */
                        }

                        /* inizializziamo occ */
                        occ=0;
                        /* per prima cosa dobbiamo aspettare la posizione
dall'altro figlio */
                        nr=read(pipe_f[2*N-i-1][0], &posLetta, sizeof(posLetta));
                        if (nr != sizeof(posLetta))
                        {
                                printf("Impossibile leggere dalla pipe per il
processo di indice %d (PRIMA LETTURA)\n", i);
                                exit(0);
                        }
                        /* printf("DEBUG- VALORE DI pos %ld per processo di indice
%d che ho ricevuto da pipe_fbis[2*N-i-1][0] %d\n", pos, i, pipe_fbis[2*N-i-1][0]);
*/
                        /* spostiamo l'I/O pointer nella posizione seguente! */
                        lseek(fd, posLetta+1, SEEK_SET);
                        while (read(fd, &ch, 1))
                        {
                                if (ch == Cz) /* se abbiamo trovato il carattere da
cercare */
                                {
                                        /* incrementiamo occ */
                                        occ++;
                                        /* calcoliamo la posizione del carattere */
                                        /* il valore ricavato dalla lseek lo
decrementiamo di 1 dato che dopo la lettura l'I/O pointer e' posizionato sul
carattere seguente quello letto */
                                        pos=lseek(fd, 0L, SEEK_CUR) - 1;
                                        /* inviamo la posizione del carattere
all'altro processo della coppia */
                                        /* printf("DEBUG- VALORE DI pos %ld per
processo di indice %d che sto per mandare su pipe_f[2*N-i-1][1] %d\n", pos, i,
pipe_f[2*N-i-1][1]); */
                                        nw=write(pipe_fbis[2*N-i-1][1], &pos,
sizeof(pos));
                                        if (nw != sizeof(pos))
                                        {
                                                printf("Impossibile scrivere sulla
pipe per il processo di indice %d\n", i);
                                                exit(0);
                                        }
                                        /* aspettiamo dall'altro processo della
coppia la nuova posizione da cui si deve riprendere la ricerca */
                                        nr=read(pipe_f[2*N-i-1][0], &posLetta,
sizeof(posLetta));
                                        if (nr != sizeof(posLetta))
                                        {
```

```
                                                /* se non mi viene inviato alcuna
posizione vuole dire che l'altro processo della coppia NON ha trovato altre
occorrenze e quindi si puÃ² terminare la lettura */
                                                break;
                                        }
                                        /* printf("DEBUG- VALORE DI pos %ld per
processo di indice %d che ho ricevuto da pipe_fbis[i][0] %d\n", pos, i,
pipe_fbis[i][0]); */
                                        /* spostiamo l'I/O pointer nella posizione
seguente! */
                                        lseek(fd, posLetta+1, SEEK_SET);
                                }
                                else
                                {  /* nulla, si continua a leggere */
                                        ;
                                }
                        }
                        exit(occ); /* torniamo il numero di occorrenze trovate
(supposto dal testo <= di 255) */
        }
}
}

/*codice del padre*/
/* chiudiamo tutte le pipe, dato che le usano solo i figli */
for (i=0;i<N;i++)
{
   close(pipe_f[i][0]);
   close(pipe_f[i][1]);
   close(pipe_fbis[i][0]);
   close(pipe_fbis[i][1]);
}

/* Attesa della terminazione dei figli */
for(i=0;i<2*N;i++)
{
   pidFiglio = wait(&status);
   if (pidFiglio < 0)
   {
      printf("Errore wait\n");
      exit(7);
   }
   if ((status & 0xFF) != 0)
               printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);
   else
   {
               ritorno=(int)((status >> 8) & 0xFF);
               printf("Il figlio con pid=%d ha ritornato %d\n", pidFiglio,
ritorno);
   }
}
exit(0);
}/* fine del main */
```

## 17 FEBBRAIO 2021

TESTO PARTE C: ATTENZIONE LEGGERE ANCHE LA NOTA SEGUENTE AL TESTO! La parte in C accetta un numero variabile di parametri Q maggiore o uguale a 2 che rappresentano nomi di file (F1, …FQ). Il processo padre deve generare un numero di processi figli pari a Q: ogni processo figlio Pq è associato ad uno dei file F1, …FQ (in ordine); la lunghezza in linee di tutti i file è uguale e non deve essere controllata. Ognuno di tali processi figli Pq esegue concorrentemente e legge tutte le linee del proprio file associato per operare il conteggio di tutti i caratteri numerici: per ogni linea, ogni processo deve riportare su standard output il numero d'ordine del processo, il suo PID, il numero di caratteri numerici della linea corrente insieme con la linea stessa. I processi figli devono sincronizzarsi a vicenda in modo che le scritture su standard output avvengano in modo ciclico dal primo processo all'ultimo fino a che si leggono linee dai singoli file. Quindi, i processi figli Pq devono usare uno schema di sincronizzazione a ring: il generico processo Pq dopo aver ricevuto l'ok dal figlio precedente stampa quanto richiesto e poi manda l'ok al figlio successivo, con PQ-1 che manda a P0. Per semplicità, il primo ciclo può essere attivato dal padre che manda un'indicazione di partenza (il primo OK) al primo figlio P0. Al termine dell'esecuzione, ogni figlio Pq ritorna al padre il numero di caratteri numerici dell'ultima linea (supposto minore di 255); il padre deve stampare su standard output il PID di ogni figlio e il valore ritornato.

NOTA BENE NEL FILE C main.c SI USI OBBLIGATORIAMENTE:

- una variabile di nome Q per il numero di processi figli;
- una variabile di nome q per l'indice dei processi figli;
- una variabile di nome linea per la linea letta correntemente dai figli dal proprio file.

**tags**: ring tra figli, i figli si scambiano più informazioni che sono un solo carattere per sincronizzarsi. (più informazioni in numero non noto)

```c
/* Soluzione della parte C del compito del 17 Febbraio 2021 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>

typedef int pipe_t[2];

int main (int argc, char **argv)
{
   int Q;                               /* numero di file/processi */
   /* ATTENZIONE NOME Q imposto dal testo! */
   int pid;                             /* pid per fork */
   pipe_t *pipes;                       /* array di pipe usate a ring da primo
figlio, a secondo figlio .... ultimo figlio e poi a primo figlio: ogni processo
legge dalla pipe q e scrive sulla pipe (q+1)%Q */
   int q,j;                             /* indici */
   /* ATTENZIONE NOME q imposto dal testo! */
   int fd;                              /* file descriptor */
   char linea[250];                     /* linea letta dai figli dal proprio file
(supponiamo bastino 250 caratteri per linea e terminatore di linea, poi tramutato
in terminatore di stringa) */
   /* ATTENZIONE NOME linea imposto dal testo! */
   int nrnum;                           /* contatore caratteri numerici per ogni
linea */
   char ok;                             /* carattere letto dai figli dalla pipe
precedente e scritta su quella successiva; N.B. NON importa quale valore sia! */
   int nr,nw;                           /* variabili per salvare valori di ritorno
```

```
di read/write da/su pipe */
   int pidFiglio, status, ritorno;      /* per valore di ritorno figli */

        /* controllo sul numero di parametri almeno 2 file */
        if (argc < 3)
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno due
parametri (file)\n", argc);
                exit(1);
        }

        Q = argc-1;     /* calcoliamo il numero di file e quindi di processi da
creare */
        printf("DEBUG-Numero di processi da creare %d\n", Q);

        /* allocazione pipe */
        if ((pipes=(pipe_t *)malloc(Q*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe\n");
                exit(2);
        }

        /* creazione pipe */
        for (q=0;q<Q;q++)
                if(pipe(pipes[q])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(3);
                }

        /* creazione figli */
        for (q=0;q<Q;q++)
        {
                if ((pid=fork())<0)
                {
                        printf("Errore creazione figlio\n");
                        exit(4);
                }

                if (pid == 0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d e sono associato al file
%s\n", getpid(), argv[q+1]);
                        /* nel caso di errore in un figlio decidiamo di ritornare
il valore -1 che sara' interpretato dal padre come 255 (valore NON ammissibile) */

                        /* chiusura pipes inutilizzate */
                        for (j=0;j<Q;j++)
                        {       /* si veda commento nella definizione dell'array
pipes per comprendere le chiusure */
                                if (j!=q)
                                        close (pipes[j][0]);
                                if (j != (q+1)%Q)
                                        close (pipes[j][1]);
                        }

                        /* apertura file */
                        if ((fd=open(argv[q+1],O_RDONLY))<0)
                        {
```

```
                                printf("Impossibile aprire il file %s\n",
argv[q+1]);
                                exit(-1);
                        }

                        /* inizializziamo l'indice dei caratteri letti per ogni
singola linea */
                        j = 0;
                        /* inizializziamo il contatore dei caratteri numerici per
ogni singola linea */
                        nrnum = 0;
                        /* con un ciclo leggiamo tutte le linee, come richiede la
specifica */
                        while(read(fd,&(linea[j]),1) != 0)
                        {
                                if (linea[j] == '\n') /* siamo a fine linea */
                                {
                                        /* dobbiamo aspettare l'ok dal figlio
precedente per scrivere: al primo giro il primo figlio riceve l'ok dal padre! */
                                        nr=read(pipes[q][0],&ok,sizeof(char));
                                        /* per sicurezza controlliamo il risultato
della lettura da pipe */
                                        if (nr != sizeof(char))
                                        {
                                                printf("Figlio %d ha letto un
numero di byte sbagliati %d\n", q, nr);
                                                exit(-1);
                                        }
                                        /* a questo punto si deve riportare su
standard output l'indice e il pid del processo, il numero di caratteri numerici
presenti e la linea letta */
                                        /* NOTA BENE: al posto dello \n dobbiamo
mettere uno \0 in modo da non avere problemi nella printf seguente: abbiamo
sovrascritto il terminatore di linea dato che comunque dopo usiamo \n nella printf!
*/
                                        linea[j]='\0';
                                        printf("Figlio con indice %d e pid %d ha
letto %d caratteri numerici nella linea %s\n", q, getpid(), nrnum, linea);
                                        /* ora si deve mandare l'OK in avanti: nota
che il valore della variabile ok non ha importanza */
                                        nw=write(pipes[(q+1)%Q]
[1],&ok,sizeof(char));
                                        /* anche in questo caso controlliamo il
risultato della scrittura */
                                        if (nw != sizeof(char))
                                        {
                                                printf("Figlio %d ha scritto un
numero di byte sbagliati %d\n", q, nw);
                                                exit(-1);
                                        }
                                        /* NOTA BENE: nell'ultima iterazione
l'ultimo figlio mandera' un OK al primo figlio che pero' non verra' ricevuto dato
che il primo figlio sara' terminato, ma questo non creera' alcun problema a patto
che il padre mantenga aperto il lato di lettura di pipes[0]: in questo modo,
l'ultimo figlio non incorrera' nel problema di scrivere su una pipe che non ha
lettori */
                                        /* si deve azzerare l'indice della linea e
il conteggio dei caratteri numerici, quest'ultimo dopo averlo salvato per poterlo
tornare correttamente, nel caso la linea corrente sia l'ultima! */
```

```
                                        j = 0;
                                        ritorno = nrnum;
                                        nrnum = 0;
                                }
                                else
                                {
                                        /* se non siamo a fine linea dobbiamo fare
il controllo sul carattere corrente */
                                        if (isdigit(linea[j])) /* se abbiamo letto
un carattere numerico incrementiamo il contatore */
                                                nrnum++;
                                        j++; /* incrementiamo sempre l'indice della
linea */
                                }
                        }
                        /* ogni figlio deve tornare il numero di caratteri numerici
dell'ultima linea */
                        exit(ritorno);
                }
        } /* fine for */

        /* codice del padre */
        /* chiusura di tutte le pipe che non usa, a parte la prima perche' il padre
deve dare il primo OK al primo figlio. N.B. Si lascia aperto sia il lato di
scrittura che viene usato (e poi in effetti chiuso) che il lato di lettura (che non
verra' usato ma serve perche' non venga inviato il segnale SIGPIPE all'ultimo
figlio che terminerebbe in modo anomalo)  */
        for(q=1;q<Q;q++) /* l'indice lo facciamo partire quindi da 1! */
        {
                close (pipes[q][0]);
                close (pipes[q][1]);
        }
        /* ora si deve mandare l'OK al primo figlio (P0): nota che il valore della
variabile ok non ha importanza */
        nw=write(pipes[0][1],&ok,sizeof(char));
        /* anche in questo caso controlliamo il risultato della scrittura */
        if (nw != sizeof(char))
        {
                printf("Padre ha scritto un numero di byte sbagliati %d\n", nw);
                exit(5);
        }

        /* ora possiamo chiudere anche il lato di scrittura, ma ATTENZIONE NON
QUELLO DI LETTURA! */
        close(pipes[0][1]);
        /* OSSERVAZIONE SU NON CHIUSURA DI pipes[0][0]: se si vuole procedere con
la chiusura di tale lato nel padre, bisognerebbe introdurre del codice ulteriore
solo nel primo figlio che vada a fare la lettura dell'ultimo OK prima di terminare!
*/

        /* Il padre aspetta i figli */
        for (q=0; q < Q; q++)
        {
                if ((pidFiglio = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(6);
                }
                if ((status & 0xFF) != 0)
```

```
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi)\n", pidFiglio, ritorno);
                }
        }

        exit(0);
}
```

# 13 LUGLIO 2022

TESTO PARTE C: ATTENZIONE LEGGERE ANCHE LA NOTA SEGUENTE AL TESTO! La parte in C accetta un numero variabile di parametri Q+2 (con Q maggiore o uguale a 2), F, L e C1, …CQ, che rappresentano rispettivamente le seguenti informazioni: il primo il nome assoluto di un file (F), il secondo la lunghezza in linee di F (L, strettamente positivo da controllare) e gli ultimi Q devono essere considerati singoli caratteri (da controllare). Il processo padre deve generare un numero di processi figli pari a Q: ogni processo figlio Pq è associato ad uno dei caratteri C1, …CQ (in ordine). Ognuno di tali processi figli Pq esegue concorrentemente e calcola il numero di occorrenze del proprio carattere associato, per ognuna delle L linee del file F. I processi padre e figli devono sincronizzarsi strettamente a vicenda utilizzando uno schema di sincronizzazione a ring che deve comprendere anche il padre in modo che, per ognuna delle L linee del file F, sullo standard output siano scritte le seguenti informazioni: il padre deve riportare il numero di linea correntemente analizzata da tutti i processi figli (NB: la numerazione delle linee deve essere fatta partire da 1), insieme con il nome del file; il primo figlio (P0) deve riportare il numero di occorrenze del proprio carattere (C1) trovate nella linea corrente, insieme con il proprio indice e il proprio PID; il secondo figlio (P1) deve riportare il numero di occorrenze del proprio carattere (C1) trovate nella linea corrente, insieme con il proprio indice e il proprio PID; così via per tutti i figli e per tutte le L linee del file F.

NOTA BENE NEL FILE C main.c SI USI OBBLIGATORIAMENTE:

- una variabile di nome Q per il numero di processi figli;
- una variabile di nome q per l'indice dei processi figli;
- una variabile di nome L per la lunghezza in linee del file F.

**tags**: ring con il padre, il padre e i figli sono inseriti in un ring e si passano informazioni in numero noto

```
/* Soluzione della parte C del compito del 13 Luglio 2022 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>

typedef int pipe_t[2];

int main (int argc, char **argv)
{
   int Q;                                /* numero di file/processi */
   /* ATTENZIONE NOME Q imposto dal testo! */
```

```c
   int pid;                             /* pid per fork */
   pipe_t *pipes;                       /* array di Q+1 pipe usate a ring da primo
figlio, a secondo figlio .... ultimo figlio e poi padre e quindi di nuovo da primo
figlio: ogni processo legge dalla pipe q e scrive sulla pipe q+1; il padre legge da
pipe Q e scrive su pipe 0! */
   int q,j;                             /* indici */
   /* ATTENZIONE NOME q imposto dal testo! */
   int L;                               /* per valore numero linee del file */
   /* ATTENZIONE NOME L imposto dal testo! */
   int fd;                              /* file descriptor */
   char ch;                             /* carattere letto dai figli dall'unico
file */
   int nrChar;                          /* contatore carattere cercato per ogni
linea */
   char ok;                             /* carattere letto dai figli dalla pipe
precedente e scritto su quella successiva: N.B. valore non importante! */
   int nr,nw;                           /* variabili per salvare i valori di
ritorno di read/write da/su pipe */
   int pidFiglio, status, ritorno;      /* per valore di ritorno figli */

        /* controllo sul numero di parametri almeno 1 file, una lunghezza e 2
caratteri */
        if (argc < 5)
        {
                printf("Numero dei parametri errato %d: ci vogliono almeno quattro
parametri (un file, la sua lunghezza in linea e 2 caratteri)\n", argc);
                exit(1);
        }

        /* ricaviamo il numero di linee del file e controlliamo che sia
strettamente maggiore di 0 */
        L = atoi(argv[2]);
        if (L <= 0)
        {
                printf("Errore %d non numero intero strettamente positivo\n", L);
                exit(2);
        }

        /* controlliamo che i parametri a partire dal terzo siano singoli parametri
*/
        for (j=3; j<argc; j++)
                if (strlen(argv[j]) !=1)
                {
                        printf("Il parametro %s NON e' un singolo carattere\n",
argv[j] );
                        exit(3);
                }

        Q = argc-3;     /* calcoliamo il numero di caratteri e quindi di processi
da creare */
        printf("DEBUG-Numero di processi da creare %d\n", Q);

        /* allocazione pipe: ATTENZIONE VANNO CREATE Q+1 pipe! */
        if ((pipes=(pipe_t *)malloc((Q+1)*sizeof(pipe_t))) == NULL)
        {
                printf("Errore allocazione pipe\n");
                exit(4);
        }
```

```
        /* creazione pipe */
        for (q=0; q<Q+1; q++)
                if(pipe(pipes[q])<0)
                {
                        printf("Errore creazione pipe\n");
                        exit(5);
                }

        /* creazione figli */
        for (q=0; q<Q; q++)
        {
                if ((pid=fork())<0)
                {
                        printf("Errore creazione figlio\n");
                        exit(6);
                }

                if (pid == 0)
                {       /* codice figlio */
                        printf("DEBUG-Sono il figlio %d e sono associato al file %s
e al carattere %c\n", getpid(), argv[1], argv[q+3][0]);
                        /* tutti i figli sono associati all'unico file */
                        /* nel caso di errore in un figlio decidiamo di ritornare
il valore -1 che sara' interpretato dal padre come 255 (valore NON ammissibile) */

                        /* chiusura pipes inutilizzate */
                        for (j=0;j<Q+1;j++)
                        {       /* si veda commento nella definizione dell'array
pipes per comprendere le chiusure */
                                if (j!=q)
                                        close (pipes[j][0]);
                                if (j != (q+1))
                                        close (pipes[j][1]);
                        }

                        /* apertura file */
                        if ((fd=open(argv[1],O_RDONLY))<0)
                        {
                                printf("Impossibile aprire il file %s\n", argv[1]);
                                exit(-1);
                        }

                        /* inizializziamo il contatore del carattere cercato per
ogni singola linea */
                        nrChar= 0;
                        /* con un ciclo leggiamo tutte le linee, come richiede la
specifica */
                        while(read(fd,&ch,1) != 0)
                        {
                                if (ch == '\n') /* siamo a fine linea */
                                {
                                        /* dobbiamo aspettare l'ok dal figlio
precedente per scrivere */
                                        nr=read(pipes[q][0],&ok,sizeof(char));
                                        /* per sicurezza controlliamo il risultato
della lettura da pipe */
                                        if (nr != sizeof(char))
                                        {
                                                printf("Figlio %d ha letto un
```

```
numero di byte sbagliati %d\n", q, nr);
                                        exit(-1);
                                }

                                /* a questo punto si deve riportare su
standard output l'indice e il pid del processo, il numero di occorrenze del proprio
carattere associato presenti nella linea corrente e il carattere associato */
                                printf("Figlio con indice %d e pid %d ha
letto %d caratteri %c nella linea corrente\n", q, getpid(), nrChar, argv[q+3][0]);
                                /* ora si deve mandare l'OK in avanti: nota
che il valore della variabile ok non ha importanza */
                                nw=write(pipes[q+1][1],&ok,sizeof(char));
                                /* anche in questo caso controlliamo il
risultato della scrittura */
                                if (nw != sizeof(char))
                                {
                                        printf("Figlio %d ha scritto un
numero di byte sbagliati %d\n", q, nw);
                                        exit(-1);
                                }
                                /* si deve azzerare il conteggio delle
occorrenze, dopo averlo salvato per poterlo tornare correttamente, nel caso la
linea corrente sia l'ultima! */
                                ritorno = nrChar;
                                nrChar = 0;
                        }
                        else
                        {
                                /* se non siamo a fine linea dobbiamo fare
il controllo sul carattere corrente */
                                if (ch == argv[q+3][0]) /* se abbiamo letto
il carattere da cercare incrementiamo il contatore */
                                        nrChar++;
                        }
                }
                /* ogni figlio deve tornare il numero di occorrenze del
proprio carattere trovate nell'ultima linea */
                exit(ritorno);
        }
} /* fine for */

/* codice del padre */
/* chiusura di tutte le pipe che non usa, a parte la prima e l'ultima */
for(q=0; q<Q+1; q++)
{
        if (q != Q)     close (pipes[q][0]);
        if (q != 0)     close (pipes[q][1]);
}
/* N.B. Poiche' in questo caso il padre e' nel ring, non ci sono problemi
di dover lasciare aperti lati di pipe che il padre non usa! */

for (j=0; j<L; j++)     /* per ogni linea del file */
{
        /* il padre deve riportare il numero di linea correntemente
analizzata dai figli, insieme con il nome del file */
        printf("Linea %d del file %s\n", j+1, argv[1]); /* il numero di
linea deve partire da 1! */
        /* il padre deve inviare un 'segnale' di sincronizzazione al
processo di indice 0 */
```

```
                nw=write(pipes[0][1],&ok,sizeof(char));
                /* anche in questo caso controlliamo il risultato della scrittura
*/
                if (nw != sizeof(char))
                {
                        printf("Padre ha scritto un numero di byte sbagliati %d\n",
nw);
                        exit(7);
                }

                /* il padre quindi deve aspettare che l'ultimo figlio gli invii il
'segnale' di sincronizzazione per fare ripartire il ring */
                nr=read(pipes[Q][0],&ok,sizeof(char));
                /* per sicurezza controlliamo il risultato della lettura da pipe */
                if (nr != sizeof(char))
                {
                        printf("Padre ha letto un numero di byte sbagliati %d\n",
nr);
                        /* andiamo comunque ad aspettare i figli */
                }
        }

        /* Il padre aspetta i figli */
        for (q=0; q < Q; q++)
        {
                if ((pidFiglio = wait(&status)) < 0)
                {
                        printf("Errore in wait\n");
                        exit(8);
                }
                if ((status & 0xFF) != 0)
                        printf("Figlio con pid %d terminato in modo anomalo\n",
pidFiglio);
                else
                {
                        ritorno=(int)((status >> 8) & 0xFF);
                        printf("Il figlio con pid=%d ha ritornato %d (se 255
problemi)\n", pidFiglio, ritorno);
                }
        }

        exit(0);
}
```